Prozessorientierte optimistisch-parallele Simulation

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum naturalium (Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
der Humboldt-Universität zu Berlin

von
Diplom-Informatiker Andreas Kunert

Präsident der Humboldt-Universität zu Berlin
Prof. Dr. Jan-Hendrik Olbertz

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II
Prof. Dr. Elmar Kulke

Gutachter:
1. Prof. Dr. Joachim Fischer
2. Prof. Dr. Bernd Page
3. Prof. Dr. Klaus Bothe

Tag der Verteidigung: 20. Dezember 2010

ABSTRACT

The main motivation of parallel discrete event-driven simulations (PDES) is a speed-up of simulation runs. This aim is reflected by two similarities shared by almost all current PDES implementations: For efficiency reasons the event-oriented modeling view is used exclusively and the implementation is done in a low-level programming language.

However, both properties impose serious constraints on the simulation model. In many cases the event-oriented modeling view – especially in contrast to the process-oriented view – prevents a structurally equivalent model of the original system. Furthermore, the actual implementation often leads to a further decomposition of the model, due to a lack of abstract concepts in the low-level programming language. The resulting implementations run efficiently, but at the price of a low comprehensibility, due to the structural difference from the original problem. Consequently, the efforts for validation or modification of the simulation model are considerably increased.

These disadvantages led to a different development route in the sequential simulation community – in which the execution time is of minor importance. Here, the combined use of the event-oriented and the process-oriented view is common practice in order to obtain a structurally equivalent and intuitive model. Moreover, the simulation model is usually implemented in a higher-level programming language, e.g. Java.

A combination of the advantages of the sequential, as well as the parallel simulation approach is desirable, especially in the case of large and complex simulation scenarios. These are only manageable if a corresponding structurally equivalent simulation model is employed. Also, the very same models will profit most from a speed-up by parallelization.

Unfortunately, the implementation of a process-oriented view in a parallel fashion is not a trivial task. Parallel simulation implementations generally suffer from additional computational cost that can offset the parallel speed-up and even lead to

a diminished parallel performance, which is even more likely to happen when using process-oriented simulation models. This is especially the case in optimistic-parallel simulations. These are characterized by the ability of the model to get into invalid states which is costly being corrected at runtime by returning the model to valid former states preliminarily saved.

The aim of this thesis is the design and implementation of a simulation library in Java. It combines the advantages of optimistic-parallel simulation (i. e. use of parallelism for speed-up) and the process-oriented modeling (creation of an intuitive and structurally equivalent model).

Another central objective of the development is to hide the internals of the optimistic-parallel simulation kernel from the modeler, in contrast to most existing PDES implementations. Instead, the implementation is encapsulated by interfaces, which resemble those of sequential simulation kernels, reducing the additional effort needed to create a suited model.

A noteworthy aspect of the implementation is the reuse of a web application framework for retroactive modification of the Java bytecode, generated by the Java compiler. This bytecode rewriting solves in an elegant way the task of realizing coroutines, which are the base for the implementation of processes as needed by process-oriented simulation models.

ZUSAMMENFASSUNG

Die Hauptmotivation bei der parallelen diskreten ereignisgesteuerten Simulation (PDES) liegt in der Realisierung möglichst schneller Simulationsläufe. Dieses Ziel spiegelt sich in zwei Gemeinsamkeiten nahezu aller derartiger Simulationsimplementationen wider: Sie stützen sich aus Effizienzgründen ausschließlich auf das ereignisorientierte Modellierungsparadigma und sind in vergleichsweise hardwarenahen Programmiersprachen umgesetzt worden.

Mit beiden Eigenschaften sind jedoch potentiell schwerwiegende Einschränkungen für den Simulationsmodellierer verbunden. Zunächst verhindert das ereignisorientierte Paradigma, insbesondere im Gegensatz zur prozessorientierten Modellierungssicht, in vielen Szenarien eine strukturäquivalente Modellierung des zu analysierenden Systems. Bei der anschließenden Implementation erzwingt nicht selten die hardwarenahe Programmiersprache durch den Mangel passender abstrakter Konzepte eine weitere Dekomposition des Simulationsmodells. Das Ergebnis eines derartigen Modellierungs- bzw. Implementationsvorganges ist häufig eine Simulationsimplementation, deren Simulationsmodell zwar sehr effizient ausführbar, dafür jedoch durch seine strukturelle Entfernung vom Ausgangssystem schlecht nachvollziehbar ist. Die durch diesen Umstand bedingte Erhöhung des Aufwandes bei der Sicherstellung der Korrektheit sowie bei späteren Modelländerungen ist beträchtlich.

Getrieben von dieser Erkenntnis folgte die Entwicklung in der nichtparallelen Simulationswelt, in der die benötigte Ausführungszeit eine sekundäre Rolle spielt, einem anderen Weg. Um ein möglichst strukturäquivalentes und damit handhabbares Simulationsmodell zu ermöglichen, ist hier inzwischen die kombinierte Nutzung von ereignis- und prozessorientierter Modellierungssicht üblich. Des Weiteren erfolgt die Implementation sequentieller Simulationen heutzutage üblicherweise in aktuellen, von der Hardware deutlich stärker abstrahierenden Programmiersprachen wie Java.

Eine Kombination der Vorteile der sequentiellen und parallelen Simulationswelten ist vor allem bei der Betrachtung großer und komplexer Szenarien wünschenswert.

Diese lassen sich meist nur bei einer adäquaten, strukturäquivalenten Modellierung beherrschen. Es sind aber auch gleichzeitig genau diese Simulationsszenarien, die am ehesten nach einer Beschleunigung der Simulationsausführung verlangen.

Eine Implementation des prozessorientierten Paradigmas in einer parallelen Simulation ist allerdings nicht unproblematisch. Mehr noch als bei ereignisorientierten Simulationsmodellen muss bei prozessorientierten darauf geachtet werden, dass die parallelisierungsbedingte Beschleunigung von Simulationsläufen nicht durch einen ebenfalls parallelisierungsbedingten, unvermeidbaren Mehraufwand zur Laufzeit zunichte gemacht wird. Dies gilt vor allem im Spezialfall der optimistisch-parallelen Simulation. Diese zeichnet sich dadurch aus, dass Simulationsmodelle während eines Simulationslaufes in ungültige Zustände geraten können, die durch eine Rückkehr des Simulationsmodells in einen früheren, korrekten Zustand korrigiert werden.

Die vorliegende Arbeit beschreibt die Konzeption und Implementation einer optimistisch-parallelen Simulationsbibliothek in Java. Diese vereint die Vorzüge optimistisch-paralleler Simulation (automatische Ausnutzung modellinhärenter Parallelität zur Simulationsbeschleunigung) mit der des prozessorientierten Paradigmas (modellabhängig strukturäquivalente und intuitive Modellbeschreibungen).

Dabei bestand ein weiteres Entwicklungsziel darin, im Gegensatz zu den meisten existierenden PDES-Implementationen die interne Arbeitsweise des optimistisch-parallelen Simulationskerns so gut wie möglich zu verbergen. Stattdessen gleichen die vom Simulationskern angebotenen Schnittstellen weitestgehend denen von sequentiellen Simulationskernen. Dadurch wird die Erstellung eines passenden Simulationsmodells im Vergleich zu anderen parallelen Simulationsimplementationen deutlich erleichtert. Auch der Mehraufwand gegenüber der Erstellung von Simulationsmodellen für rein sequentielle Simulationskerne ist relativ gering.

Ein besonderer Aspekt der Implementation ist die zweckentfremdete Verwendung eines Webapplikationsframeworks, welches durch die nachträgliche Modifikation des vom Java-Compiler generierten Java-Bytecodes die Verwendung des in Java ansonsten unbekannten abstrakten Konzeptes einer Coroutine ermöglicht. Dadurch konnte eine elegante Möglichkeit der Implementation von Simulationsprozessen realisiert werden, die für die Umsetzung von prozessorientierten Simulationsmodellen benötigt wird.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

QUELLTEXTVERZEICHNIS

KAPITEL 1

EINLEITUNG

1.1 Problemstellung/Motivation

Das Forschungsgebiet der Computersimulation kann mit seinen Ursprüngen in den 1940er Jahren durchaus als eines der ältesten innerhalb der Informatik bezeichnet werden. Das Teilgebiet der diskreten ereignisgesteuerten Simulation (DES) ist nur unwesentlich jünger; die frühesten Arbeiten zu diesem Thema sind in den 1950er Jahren entstanden.

Demgegenüber erscheint die Disziplin der parallelen diskreten ereignisgesteuerten Simulation (PDES) als eine geradezu junge Richtung. Dennoch wurden auf diesem Gebiet in den ca. 25 Jahren seines Bestehens große Fortschritte erzielt, wobei die zugrundeliegende Motivation stets in einer erhofften Beschleunigung von Simulationsausführungen bestand und besteht.

Dabei sind Simulationsbeschleunigungen aus verschiedenen Gründen interessant. So ermöglichen sie die Betrachtung einer größeren (Modell-)Zeitspanne und/oder eines größeren Parameterraumes während eines Simulationslaufes. Alternativ oder zusätzlich kann eine Beschleunigung auch zur Verarbeitung größerer und/oder komplexerer Simulationsmodelle verwendet werden.

Aber auch ohne Modifikationen an Simulationsmodell oder Experimentkonfiguration kann sich eine Simulationsbeschleunigung als nützlich erweisen. Dies ist der Fall, wenn im Rahmen einer Experimentserie erst durch die Beschleunigung eine hinreichend große Anzahl von Simulationsläufen ermöglicht wird, die für eine statistisch relevante Anzahl von Simulationsergebnissen nötig ist.

Mangelnde Akzeptanz paralleler Simulationsalgorithmen

Doch trotz der erreichten Erfolge spielen parallele Simulationsalgorithmen in der Praxis bis heute keine relevante Rolle. Diese Feststellung wurde auch von der PDES-

Forschergemeinde getroffen und kontrovers diskutiert. Die Bedeutung der allgemeinen Akzeptanz für das Forschungsgebiet der parallelen Simulation spiegelt sich in den Titeln der beiden meistzitierten Veröffentlichungen zu dieser Diskussion wider: „PDES: Will the Field Survive?“ von Fujimoto [Fuj93] und „PDES: So Who Cares?“ von Nicol [Nic97].

Ein wesentlicher Grund für die Ablehnung der parallelen Simulation in der Praxis besteht dabei, insbesondere im Vergleich zur sequentiellen Simulation, in einer zu geringen logischen Trennung von Simulationsmodell und Ausführungsalgorithmik. Nahezu alle existierenden PDES-Implementationen verlangen bei der Erstellung passender Simulationsmodelle ein vollständiges Verständnis der inneren Abläufe des später parallel arbeitenden Simulationskerns. In der Einleitung von [BB97] wird diese Problematik wie folgt beschrieben:

> “Existing PDES systems tend to expose the underlying synchronization protocols and either expect or require simulationists to understand their intricacies; meanwhile, important modelling issues such as the choice of world view are either ignored altogether or compromised as the result of implementational expediency.”

Modellierungssichten

Die Grundlage jeder Simulation besteht in einem Simulationsmodell, das das zu simulierende Szenario problemadäquat repräsentiert. Die Grundlage für ein solches Simulationsmodell bildet wiederum die Modellierungssicht, mittels derer das Modell erstellt wird. Dabei haben sich in der Praxis zwei Modellierungssichten durchgesetzt: die ereignisorientierte und die prozessorientierte.

Prinzipiell sind beide Modellierungssichten zur Erstellung problemadäquater Simulationsmodelle geeignet. Allerdings ermöglicht in vielen Szenarien ausschließlich die stärker abstrahierende, prozessorientierte Sicht ein Simulationsmodell, das zum jeweiligen Ausgangsszenario auch strukturell äquivalent ist. Derartige Simulationsmodelle sind in der Regel deutlich nachvollziehbarer, was sich nicht nur während der Entwicklung inklusive Fehlersuche und -beseitigung, sondern auch bei späteren Modelländerungen positiv bemerkbar macht. Des Weiteren lassen sich Simulationsergebnisse bei strukturäquivalenten Simulationsmodellen wesentlich leichter interpretieren.

Allerdings ist die Verwendung prozessorientierter Simulationsmodelle gegenüber ereignisorientierten auch mit Nachteilen verbunden. Um den höheren Grad an Abstraktion im Simulationsmodell unterstützen zu können, muss im Simulationskern zur Laufzeit ein größerer algorithmischer Aufwand betrieben werden. Dementsprechend ist die Implementation eines solchen Simulationskerns anspruchsvoller. Nicht weniger gravierend sind die Auswirkungen bezüglich der Ausführungsgeschwindigkeit während eines Simulationslaufes. Bedingt durch den höheren Aufwand werden prozessorientierte Simulationsmodelle prinzipiell langsamer ausgeführt als semantisch äquivalente ereignisorientierte.

Sowohl die Nachteile zur Laufzeit als auch der deutlich erhöhte Implementationsaufwand sind dafür verantwortlich, dass die prozessorientierte Modellierungssicht in der parallelen Simulationswelt nie eine wesentliche Rolle gespielt hat. Zum einen widerspricht eine prinzipiell langsamere Simulationsausführung dem zentralen Ziel paralleler Simulationsalgorithmen, zum anderen ist die Implementation eines parallelen Simulationskerns schon ohne die erweiterten Anforderungen prozessorientierter Modellierungskonzepte kompliziert genug. Infolgedessen unterstützen nahezu alle existierenden PDES-Implementationen ausschließlich ereignisorientierte Simulationsmodelle, auch wenn die Arbeit des Simulationsmodellierers darunter leidet. In [BB97] wird diese Situation folgendermaßen zusammengefasst:

> "In marked contrast, almost all PDES research has been based on the event scheduling world view. Although this can make simulators, protocols, etc., easier to develop, it has the drawback that simulation models are then harder to write."

Konservativ- und optimistisch-parallele Simulationsalgorithmen

Prinzipiell lassen sich alle parallelen Simulationsalgorithmen einer der beiden Klassen *konservativ* oder *optimistisch* zuordnen. Die zentrale Eigenschaft konservativ-paralleler Simulationsalgorithmen besteht darin, dass, analog zu sequentiellen Simulationen, das Simulationsmodell während eines Simulationslaufes niemals einen ungültigen Zustand annehmen kann. Diese Zusicherung gilt ungeachtet der Tatsache, dass in einer parallelen Simulation sowohl räumlich als auch zeitlich (bzgl. Modellzeit) getrennte Teile des Simulationsmodells gleichzeitig (bzgl. Ausführungszeit) bearbeitet werden. Damit diese Zusicherung uneingeschränkt gilt, wird in konservativ-parallelen Simulationsalgorithmen vor jedem Berechnungsschritt auf jeder der parallel arbeitenden Recheneinheiten überprüft, ob durch diesen die Gültigkeit des Simulationsmodells gefährdet ist. Kann die Gültigkeit nicht garantiert werden, wird mit dem entsprechenden Berechnungsschritt gewartet, bis durch die Arbeit anderer Recheneinheiten die Voraussetzungen für eine gefahrlose Ausführung gegeben sind.

Dabei besteht das Problem darin, dass die meisten Simulationsmodelle zwar durchaus räumlich und/oder zeitlich voneinander unabhängige Bestandteile aufweisen, die problemlos parallel verarbeitet werden könnten. Allerdings sind konservativ-parallele Simulationsalgorithmen meistens nicht in der Lage, diese auch aufzuspüren und auszunutzen. Als Ausweg wird bei den meisten konservativ-parallelen Simulationen der Ansatz gewählt, das Simulationsmodell zusätzlich um Informationen bezüglich der räumlichen und zeitlichen Kopplung einzelner Modellbestandteile zu erweitern. Da bis auf diese zusätzlichen Informationen konservativ-parallele Simulationsmodelle weitestgehend sequentiellen Modellen ähneln, ist ihre Erstellung relativ unkompliziert, insbesondere im Vergleich zu optimistisch-parallelen Simulationsmodellen.

Bei optimistisch-parallelen Simulationsalgorithmen tritt das Problem der Erkennung gefahrlos parallel verarbeitbarer Modellbestandteile hingegen gar nicht auf.

Der Grund dafür liegt darin, dass das Simulationsmodell in optimistisch-parallelen Simulationen während eines Simulationslaufes prinzipiell ungültige Modellzustände annehmen darf. Auf dieser Basis können alle verfügbaren Recheneinheiten stets voll ausgelastet parallel auf verschiedenen Teilen des Simulationsmodells arbeiten. Um dennoch zum Ende einer Simulation verwertbare gültige Ergebnisse vorweisen zu können, müssen jedoch ungültige Zustände eines Simulationsmodellteils noch während der laufenden Simulation korrigiert werden.

Dies geschieht durch eine Rückführung des betroffenen Modellteils in einen früheren, gültigen Zustand. Da sich dieser Modellteil dadurch jedoch in der Modellzeit rückwärts bewegt, treten mehrere neue Probleme auf, die teilweise nur durch Einschränkungen des Simulationsmodells gelöst werden können. Infolgedessen unterstützen die meisten existierenden optimistisch-parallelen Simulationskerne nur Simulationsmodelle, die auf Basis einer deutlich restriktiveren Modellierungssicht erstellt wurden.

Um dennoch abstraktere Simulationsmodelle, die weitestgehend sequentiellen und im Idealfall sogar prozessorientiert erstellten Simulationsmodellen ähneln, ausführen zu können, wäre eine Abstraktionsschicht zwischen diesen Modellen und dem Simulationskern nötig. Da diese jedoch nicht nur die benötigte Simulationslaufzeit nachteilig beeinflusst, sondern vor allem sehr aufwendig in der Implementation ist, verzichten nahezu alle existierenden optimistisch-parallelen Simulationsimplementation auf selbige. In generischen Simulationsbibliotheken zur Erstellung eigener Simulationen ist eine solche Abstraktionsschicht bisher gar nicht anzutreffen.

Parallelisierung jenseits der Simulation

Es sollte nicht unerwähnt bleiben, dass zurzeit auch jenseits des Gebietes der Simulation ein starker Trend zur Parallelisierung von Algorithmen existiert. Befeuert wurde dieser vor allem durch einen Wechsel in der zentralen Strategie der Prozessorentwicklung in den vergangenen Jahren. Während bis vor ca. fünf Jahren die wirksamste Maßnahme der Prozessorbeschleunigung in einer Erhöhung der Taktfrequenz bestand, stieß dieser Ansatz bei etwa 4 GHz an physikalische (vor allem thermische) Grenzen. Um dennoch die Prozessorgeschwindigkeit weiter zu erhöhen, wurde nach alternativen Ansätzen gesucht und ein solcher wurde schließlich in der parallelen Ausführung der Prozessoraufgaben gefunden [Sut05].[1]

Besonders interessant ist dabei die Entwicklung ab 2004, wo erstmals normale PC-Prozessoren mit mehreren Prozessorkernen vorgestellt wurden. Rechner, die auf solchen Prozessoren basieren, stellen sich gegenüber der darauf laufenden Software wie echte Mehrprozessorsysteme dar [Hül06]. Zum Zeitpunkt der Beendigung der vorliegenden Arbeit (Mitte 2010) besitzen die meisten am Markt verfügbaren Desktop-Prozessoren zwei bis vier, in Einzelfällen bis sechs Prozessorkerne, während im (x86-)Serverbereich bereits Prozessoren mit zwölf Kernen erhältlich sind.

[1] Parallelrechner sind zwar bei weitem keine neue Entwicklung, allerdings waren sie bis dahin nahezu ausschließlich im Serverbereich anzutreffen.

Verbunden mit der Erkenntnis, dass in näherer Zukunft auf keinem anderen Weg größere Sprünge in der Prozessorentwicklung zu erwarten sind, wurde die Parallelisierung auch auf Softwareebene verstärkt berücksichtigt. Es besteht nämlich ein zentrales Problem darin, dass nicht speziell für eine parallele Ausführung entwickelte Software auch auf Mehrprozessorsystemen in der Regel nur von einem einzelnen Prozessor ausgeführt werden kann, so dass verfügbare Rechenleistung ungenutzt bleibt [Gle10, Sut05].

Es ist insbesondere festzuhalten, dass die zahlreich vorhandenen sequentiellen Programmimplementationen nicht von der derzeitigen Prozessorentwicklung, die primär auf der Erhöhung der Anzahl integrierter Prozessorkerne basiert, profitieren. Vorher galt, dass jede Prozessorbeschleunigung automatisch zu einer Beschleunigung der bereits implementierten Software führte. Dieser Umstand wurde auch im Simulationsbereich ausgenutzt, vor allem zur nachträglichen und erst durch die Beschleunigung praktikablen Erweiterung des Simulationsmodells in bereits bestehenden Simulationsimplementationen.

Im zu dieser Problematik häufig zitierten Artikel “The Free lunch is over” [Sut05] wird diese Ausgangssituation folgendermaßen beschrieben:

> “Most classes of applications have enjoyed free and regular performance gains for several decades, even without releasing new versions or doing anything special, because the CPU manufacturers [. . .] have reliably enabled ever-newer and ever-faster mainstream systems.”

Seit der Konzentration der Prozessorentwicklung auf den Schwerpunkt Parallelisierung gilt diese Aussage nur noch für eine Teilmenge der implementierten Programme und zwar genau diejenigen, bei deren Entwicklung die (potentiell) parallele Ausführung bereits berücksichtigt wurde. Implementationen diskreter ereignisgesteuerter Simulationen werden darunter nur im Ausnahmefall zu finden sein. Der Grund dafür liegt in ihrer prinzipiellen Arbeitsweise, die auf der streng sequentiellen Abarbeitung von Ereignissen basiert, die wiederum in einer zentralen Datenstruktur gespeichert sind.

Die Konsequenz aus der Abkopplung der Software- von der Prozessorbeschleunigung ist ebenfalls in [Sut05] beschrieben:

> “Starting today, the performance lunch isn’t free anymore. [. . .] If you want your applications to benefit from the continued exponential throughput advances in new processors, it will need to be a well-written *concurrent* (usually multithreaded) application.”

Vor diesem Hintergrund erscheint es mehr als sinnvoll, Simulationsalgorithmen und -implementationen gezielt für die Verwendung auf parallel arbeitenden Rechnern zu entwickeln. Zum einen gibt es keinen anderen Weg, die bereits jetzt in parallel arbeitenden Prozessoren verfügbare Rechenleistung auszunutzen und zum anderen ist nur so das weitere Profitieren von zukünftigen Prozessorweiterentwicklungen möglich.

1.2 Ziel der Arbeit

Trotz der genannten Schwierigkeiten ist der Versuch der Implementation einer optimistisch-parallelen Simulationsbibliothek, deren Simulationskern prozessorientierte Simulationsmodelle unterstützt, lohnenswert. Im Idealfall vereinigt eine solche Bibliothek die Vorteile der optimistisch-parallelen Simulation (parallele Ausführung voneinander unabhängiger Modellbestandteile *ohne* vorherige Markierung dieser im Simulationsmodell) mit denen der prozessorientierten Modellierung (nachvollziehbare, strukturäquivalente und dadurch gut handhabbare Simulationsmodelle).

Im Rahmen dieser Arbeit wurde eine solche Simulationsbibliothek entworfen und anschließend prototypisch implementiert. Dabei bestand ein Ziel darin, die inneren Abläufe des Simulationskerns so weit wie möglich vor dem Simulationsmodell zu verbergen. Stattdessen wurde versucht, die Schnittstellen des Simulationskerns denen von sequentiellen Simulationskernen anzupassen. Dadurch gleichen die vom Simulationskern akzeptierten Simulationsmodelle weitestgehend sequentiellen Modellen, was den genannten Zielen bezüglich einfacher Modellierung entspricht.

Eine besondere Problemstellung ergab sich erst während der Implementation. Zur Umsetzung der Prozesse prozessorientierter Simulationsmodelle wird zwingend das Programmierkonzept einer Coroutine benötigt. Diese werden jedoch von der gewählten Implementationssprache Java nicht direkt unterstützt. Auch übliche Java-Coroutinen-Implementationsansätze Dritter, die teilweise sogar aus dem Bereich der (sequentiellen) Simulation kommen, stellen keine Lösung dar, da die Coroutinen in einem optimistisch-parallelen Simulationskern zusätzlich die Eigenschaft aufweisen müssen, sich unter Umständen mehrmals von einem früheren Zustand aus fortsetzen zu lassen.

Zur Lösung dieses Problems wurden zunächst verschiedene Klassen von Coroutinen und deren Umsetzung in verschiedenen Programmiersprachen betrachtet. Das mit Coroutinen eng verwandte Konzept der Continuations wurde analog analysiert und schließlich wurden die zentralen Eigenschaften ermittelt, die eine Coroutinen/Continuation-Implementation zur Umsetzung von Prozessen in optimistisch-parallelen Simulationskernen aufweisen muss.

Mit dem Ziel, eine solche Implementation nicht vollständig selbst erstellen zu müssen, wurden existierende generische Coroutinen/Continuation-Implementationen auf die ermittelten Eigenschaften überprüft. Schließlich wurde eine verwendbare Umsetzung im Framework JAVAFLOW identifiziert, dessen Ursprünge im simulationsfernen Gebiet der Webapplikationsentwicklung liegen. Dank der zweckentfremdeten Verwendung diese Frameworks konnte die Implementation der avisierten Simulationsbibliothek erfolgreich abgeschlossen werden. Anhand mehrerer Beispielsimulationen wurde anschließend überprüft, wie sehr diese Simulationsbibliothek und der darin enthaltene Simulationskern die an sie gerichteten Erwartungen bezüglich hoher Ausführungsgeschwindigkeit bei gleichzeitig intuitiver Modellierung erfüllen.

Eine Skizzierung des gesamten Implementationsvorhabens wurde mit dem

Schwerpunkt der Realisierung der Prozessimplementation im Rahmen der Middle Eastern Simulation Multiconference 2008 veröffentlicht [Kun08].

1.3 Verwandte Arbeiten

Es gibt zahlreiche Veröffentlichungen über universell einsetzbare Simulationsbibliotheken, die prozessorientiert erstellte Simulationsmodelle unterstützen. Als früheste derartige Bibliothek gilt DEMOS [Bir81], das 1979 von Graham Birtwhistle in der Programmiersprache SIMULA[2] entwickelt wurde. DEMOS gilt als Meilenstein in der Simulationsentwicklung und hat die Entwicklung vieler späterer prozessorientierter Simulationsbibliotheken maßgeblich beeinflusst. Unter aktuellen Simulationsbibliotheken, die sowohl in ihrer Programmierschnittstelle als auch im Simulationskern deutlich sichtbare Spuren von DEMOS aufweisen, befinden sich DESMO-J [LP99], JAVASIMULATION [Hel00], JDISCO [Hel01], ODEM [FA96] und ODEMX [Ger03]. Da diese Simulationsbibliotheken in der vorliegenden Arbeit zu Vergleichszwecken bezüglich der Prozessimplementation verwendet werden, befindet sich in Anhang B zu jeder jeweils noch eine detailliertere Vorstellung.

Die Simulationskerne aller genannten Simulationsbibliotheken arbeiten ausschließlich sequentiell. Es existieren nur wenige Simulationsbibliotheken, deren Kerne die parallele Ausführung prozessorientierter Simulationsmodelle unterstützen. Dabei handelt es sich jedoch jeweils ausschließlich um konservativ-parallele. Beispielbibliotheken dafür sind DSOL [JLV02] und SPADES/JAVA [TNO02, TN02].

Im Bereich der optimistisch-parallelen Simulation existieren zwar einige universelle Simulationsbibliotheken, die aber alle auf der Basis von ereignisorientierten Simulationsmodellen arbeiten. Als frühester Vertreter gilt hier sicherlich das TIME WARP OPERATING SYSTEM, in dessen Rahmen das erste Mal das Konzept der optimistisch-parallelen Simulation sowie der Begriff des Time-Warps vorgestellt wurden.

Eine Simulationsimplementation, die zwar keine universelle Simulationsbibliothek darstellt, dafür aber wirklich ein prozessorientiertes Simulationsmodell mit einer optimistisch-parallelen Ausführung kombiniert, ist TED [PF98]. Bei TED handelt es sich um einen Simulator für Telekommunikationsnetzwerke, die mit Hilfe einer domänenspezifischen Sprache spezifiziert werden. Dabei werden die einzelnen aktiven Netzwerkkomponenten in der Ausführung durch Prozesse realisiert. Zur generischen Simulationsimplementation ist TED zwar nicht geeignet, da die Prozessimplementation einige durch die eingegrenzte Domäne gegebene Modellvereinfachungen voraussetzt. Dennoch demonstriert die Simulationsimplementation beeindruckend, wie die optimistisch-parallele Simulation die Verwendung sehr großer und komplexer Simulationsmodelle unterstützt bzw. erst ermöglicht: In einzelnen Testläufen wurden erfolgreich Simulationsmodelle mit mehr als einer Million Prozessen ausgeführt.

Wie später in der Arbeit noch detailliert ausgeführt wird, stammt die Idee, Java-Programme durch Bytecode-Manipulation um das Konzept der Coroutinen/Conti-

[2] Je nach Quelle eine der ersten oder aber *die* erste objektorientierte Programmiersprache überhaupt.

nuations zu erweitern, aus dem Umfeld der Webapplikationsentwicklung. An dieser Stelle sind insbesondere die ebenfalls später noch näher betrachteten Projekte JAVAFLOW [Apac] und RIFE/CONTINUATIONS [Bev08, RIF] zu nennen.

Auch im Simulationsumfeld wurde die nachträgliche Bytecode-Modifikation bereits erfolgreich angewendet und zwar im Projekt JIST [Bar04]. Das Ziel war hier jedoch nicht die Implementation von universellen Coroutinen zur Prozessimplementation, sondern die besonders effiziente Umsetzung eines sequentiellen[3] Simulationskerns. Dieses Ziel hat JIST durchaus erreicht – bei nahezu allen semantisch äquivalenten Simulationsmodellen ist ein Simulationslauf bei JIST schneller als bei jedem anderen, derzeit verfügbaren Simulationskern auf Java-Basis. Allerdings ist die schnelle Simulationsausführung mit besonders starken Restriktionen bei der Modellerstellung verbunden. Auch JIST wird in Anhang B etwas näher vorgestellt.

Wie beschrieben, existierte bisher keine universell einsetzbare Simulationsbibliothek, die gleichzeitig sowohl prozessorientierte Simulationsmodelle als auch eine optimistisch-parallele Ausführung derselbigen unterstützt. Insofern wurde bei der Implementation der im Rahmen dieser Arbeit entstandenen Simulationsbibliothek Neuland betreten, auch wenn die einzelnen Teilaspekte durchaus gut erforscht sind.

1.4 Aufbau der Arbeit

Im unmittelbar anschließenden Kapitel werden die für das Verständnis der Arbeit benötigten, allgemeinen Simulationsgrundlagen vorgestellt. Auf diesen aufbauend, erfolgt in Kapitel 3 eine Einführung in die Grundlagen der parallelen, ereignisgesteuerten Simulation.

Die beiden darauffolgenden Kapitel bilden den Kern der Arbeit: In Kapitel 4 werden zunächst die theoretischen Grundlagen zur Umsetzung von Simulationsprozessen in optimistisch-parallelen Simulatoren sowie verschiedene Implementationsansätze diskutiert. In Kapitel 5 erfolgt dann eine ausführliche Beschreibung der Implementation der im Rahmen dieser Arbeit entstandenen Simulationsbibliothek MYTIMEWARP.

Kapitel 6 widmet sich zunächst der Beschreibung eines mit Hilfe von MYTIMEWARP implementierten Simulationsmodells. Anschließend werden im selben Kapitel verschiedene auf Basis dieses Simulationsmodells durchgeführte Experimentserien vorgestellt und die dabei ermittelten Ergebnisse ausführlich diskutiert.

Den Abschluss der Arbeit bilden schließlich die Zusammenfassung sowie ein Ausblick in Kapitel 7.

[3] Eine Erweiterung des Simulationskerns um parallele Ausführungsmechanismen war zwar seitens der Entwickler angedacht, wurde aber nie umgesetzt.

KAPITEL 2

SIMULATIONSGRUNDLAGEN

Dieses Kapitel beschäftigt sich mit den theoretischen und simulationsspezifischen Grundlagen der vorliegenden Arbeit. Im Folgenden wird zunächst die Simulation als experimentelles Analyseverfahren motiviert. Anschließend werden Grundbegriffe der Simulation vorgestellt, mit denen eine genauere Simulationsdefinition möglich ist, sowie verschiedene Klassifikationen für Simulationen diskutiert. Am Ende des Kapitels wird ein Ansatz zur Implementation von diskreten, ereignisgesteuerten Simulationen vorgestellt.

2.1 Motivation

Die Hauptmotivation hinter jeder Simulation ist die zielgerichtete Analyse eines Phänomens, das sich nicht oder nicht ausschließlich mit anderen Analyseverfahren behandeln lässt. Die Gründe, die dabei alternative Analyseverfahren, insbesondere direkte Experimente am Phänomen selbst, ausschließen, sind vielfältig und treten selten einzeln auf. Beispiele für derartige Ausschlussgründe sind:

- das Phänomen ist fiktiv und (noch) nicht existent, was ein Experiment an selbigem ausschließt (z. B. Produktionsanlage in der Planungsphase),
- das Phänomen ist zu komplex, als dass es formal beschrieben werden könnte, bzw. die formale Beschreibung ist zu komplex, um formal analysiert werden zu können (z. B. Vorhersage von Naturkatastrophen),
- das Phänomen entwickelt sich zu langsam für ein Experiment, was in der Regel auch gleichzeitig dessen Wiederholbarkeit ausschließt (z. B. Betrachtungen in der Plattentektonik),
- ein Experiment am Phänomen ist zu aufwendig/kostenintensiv, um hinreichend oft wiederholt werden zu können (z. B. Crashtests von Fahrzeugen),

- ein Experiment am Phänomen ist zu aufwendig/kostenintensiv, um überhaupt durchgeführt werden zu können (z. B. Crashtests von Flugzeugen) oder
- ein Experiment am Phänomen ist (potentiell) zu gefährlich (z. B. Optimierung von Arbeitsabläufen in einem Atomkraftwerk).

Das Analyseverfahren der Simulation ermöglicht dennoch eine zumindest partielle Analyse dieser Phänomene durch eine Vereinfachung der ursprünglichen Problemstellung. Im Folgenden werden die Vorgehensweise und die dabei benötigten Grundkonzepte der Simulation erläutert.

2.2 Grundbegriffe

System

Der erste Schritt einer Simulation besteht in der Identifikation eines sogenannten *Systems*. Dieses stellt eine erste formale Repräsentation des Ausgangsphänomens dar.

In der Simulationsliteratur existieren verschiedene Grundauffassungen des Systembegriffs. Eine der einfachsten Systemdefinitionen stammt von Gaines [Gai80]. Nach ihm ist jeder wohldefinierte, vom Rest des Universums abgegrenzte Teil ein System, d. h., ein System wird definiert durch die Menge aller seiner Bestandteile, genannt *Entitäten* (*entities*), und die Beziehungen dieser zu der Außenwelt. Dieser Systembegriff ist rekursiv anwendbar, d. h., jeder wohldefinierte Teil eines Systems ist wieder ein System. Die wohl kürzeste Zusammenfassung der Definition von Gaines bildet sein Zitat:

> "A system is what is distinguished as a system."

Auch die Systemdefinition von Ashby [Ash56] basiert auf der Systemidentifikation durch Herauslösung einer Teilmenge. Zusätzlich zu Gaines fordert Ashby jedoch eine Reduktion auf ausschließlich diejenigen Systemelemente und Abhängigkeiten, die bezüglich der am System durchzuführenden Untersuchung unbedingt benötigt werden:

> "At this point, we must be clear about how a 'system' is to be defined. Our first impulse is to point at the pendulum and to say 'the system is that thing there'. This method, however, has a fundamental disadvantage: every material object contains no less than an infinity of variables, and therefore, of possible systems. [...] Any suggestion that we should study 'all' the facts is unrealistic, and actually the attempt is never made. What is necessary is that we should pick out and study the facts that are relevant to some main interest, that is already given."

Cellier, der sich in [Cel91] mit beiden vorhergehenden Definitionen auseinandergesetzt hat, fügt diesen noch zusätzlich die Kontrollierbarkeit und Beobachtbarkeit

eines Systems hinzu. Ein System muss demnach eine Möglichkeit zur Akzeptanz von Eingaben besitzen und auf diese mit konkreten Ausgaben reagieren. Damit einher geht die Forderung, dass sowohl Ein- als auch Ausgaben des Systems messbar sind:

> "Another property of a 'system' is the fact that it can be 'controlled' and 'observed'. [...] In general, we ought to be able to assign values to at least some of the 'inputs' of the system, and observe the behaviour of the system by recording the resulting 'output'."

Von Karplus, dessen Systembegriff mit dem von Cellier äquivalent ist, wurden in [Kar77] die wesentlichen Anforderungen bei der Systembildung beschrieben, aus denen sich die Grundeigenschaften eines Systems ableiten lassen:

- **Separabilität**
  Ein System besitzt eine wohl definierte Systemgrenze, die es als eigenständige Einheit definiert. Jedes Element, das keine Entität des Systems darstellt, gehört zur *Systemumgebung* (*system environment*).
- **Selektivität**
  Sobald die Systemgrenzen festgelegt wurden, erfolgt eine Selektion: Unter allen möglichen Interaktionen/Abhängigkeiten über die Systemgrenze hinweg (also zwischen Entitäten des Systems und Elementen der Umwelt) werden nur diejenigen bei der Systembildung berücksichtigt, die für die zu analysierende Problemstellung relevant sind.
- **Kausalität**
  Die Ausgaben des Systems (aus Sicht der Umwelt) sind ausschließlich abhängig von den Eingaben sowie einem inneren Systemzustand. Die Eingaben dürfen hingegen nie von den Ausgaben beeinflusst werden (sollte dies der Fall sein, sind die Systemgrenzen zu eng gesetzt worden).

In dieser Arbeit wird der Systembegriff von Cellier bzw. Karplus als Grundlage für alle weiteren Betrachtungen verwendet.

Experiment

Nach Cellier ist ein *Experiment* die Beobachtung der Ausgaben eines Systems, während dieses mit Eingaben versehen wird:

> "An experiment is the process of extracting data from a system by exerting it through its inputs."

Wie bereits in Abschnitt 2.1 beschrieben, existieren zahlreiche Gründe, die ein Experiment am System verhindern, selbst wenn das System bereits eine Abstraktion des originalen Phänomens darstellt. Eine Simulation umgeht dieses Problem durch die Einführung einer zusätzlichen Abstraktionsebene in Form eines (Simulations-) Modells.

Modell

Eine der prägnantesten Definitionen für ein *Modell* stammt von Minsky [Min65]:

> "A model (M) for a system (S) and an experiment (E) is anything to which E can be applied in order to answer questions about S."

Ein zentraler Punkt dieser Definition ist die Tatsache, dass Modelle experimentabhängig sind, d. h., es gibt keine „besten" generischen Modelle für ein System, sondern jeweils nur für ein konkretes Experiment. Dieses Detail bildet die Grundlage dafür, dass auch von sehr komplexen Systemen experimentabhängig einfache, handhabbare Modelle erstellt werden können. Eng damit verbunden ist aber auch eine der größten Fehlerquellen im Bereich der Simulation: die Verwendung ungeeigneter Modelle, also von Modellen, bei deren Erstellung experimentnotwendige Details unberücksichtigt blieben.

Erwähnenswert bei der Definition von Minsky ist, dass das definierte Modell selbst wieder ein System darstellt. Eine Modellbildung kann also iterativ erfolgen, wobei jedes Modell/System eine Abstraktion eines konkreteren Modells/Systems darstellt.

Simulation

Mit den Definitionen von Modell und Experiment ist eine kurze und prägnante Simulationsdefinition möglich. Von Korn und Wait [KW78] stammt die folgende:

> "A simulation is an experiment performed on a model."

Werden die notwendigen Vor- und Nacharbeiten einer Simulation mit einbezogen, besteht das gesamte Analyseverfahren der Simulation aus den folgenden drei Schritten:

1. Ein Simulationsmodell des Phänomens wird erstellt. Dabei werden nur die Aspekte des Phänomens berücksichtigt, die für die zu lösende Problemstellung wesentlich sind, während von problemfernen Aspekten abstrahiert wird.
2. Anstatt das Modell formal zu analysieren, wird es im Rahmen eines Experimentes vorher definierten Eingaben ausgesetzt und seine Reaktion darauf beobachtet.
3. Die beobachteten Ergebnisse am Modell werden rückinterpretiert auf das Originalphänomen.

Computersimulation

Während die vorhergehende Simulationsdefinition (analog zu den genannten Definitionen von Modell und Experiment) Simulationen in den verschiedensten wissenschaftlichen Disziplinen umfasst, wird sich die vorliegende Arbeit ausschließlich mit dem Spezialfall der *Computersimulation* beschäftigen.

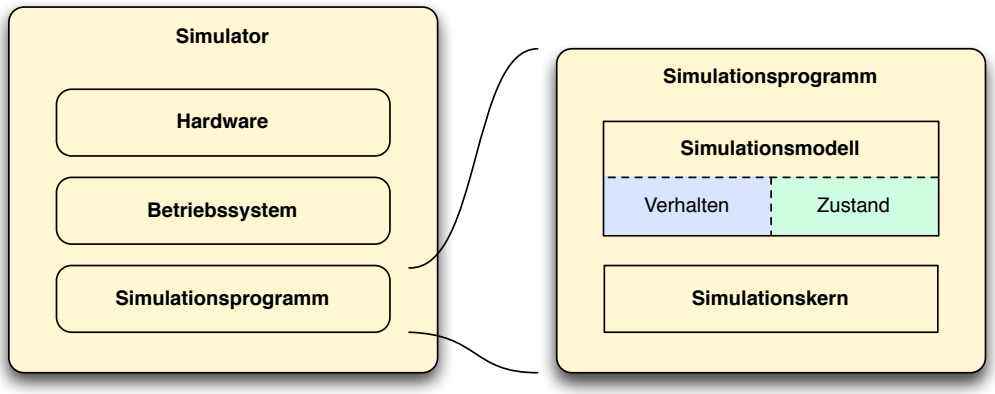

Abbildung 2.1: Schematischer Aufbau eines Simulators

Die Besonderheit einer Computersimulation besteht darin, dass das Simulationsmodell in Form eines Programmes bzw. Programmteils (wahlweise als Quelltext oder als Binärdatei) vorliegt. Dieses Programm besitzt eine Menge von *Zustandsvariablen* (*state variables*), deren Belegungen während der Laufzeit des Programmes in ihrer Gesamtheit den Systemzustand repräsentieren. Außerdem enthält das Programm eine Menge von sogenannten *Verhaltensmethoden*, die zur Laufzeit das Verhalten des Systems über die Zeit bzw. auf Einflüsse aus der Systemumgebung nachbilden.

Um ein ausführbares *Simulationsprogramm* zu erhalten, müssen dem programmatischen Simulationsmodell noch mindestens die beiden folgenden Funktionalitäten hinzugefügt werden:

- Herstellung eines initialen Modellzustandes, d. h. Belegung aller Zustandsvariablen mit Initialwerten und
- Ansteuerung der Verhaltensmethoden zur Nachbildung des Verhaltens des Originalsystems über die Zeit.

Obwohl es prinzipiell möglich wäre, sowohl das Simulationsmodell als auch die Ansteuerung desselbigen gemeinsam zu implementieren, hat es sich als praktischer erwiesen, letztere in eine separate Programmkomponente, den *Simulationskern* (*simulation kernel*) auszulagern. Dies ist darin begründet, dass die Ausführungsalgorithmik prinzipiell für alle Simulationsmodelle identisch ist. Infolgedessen muss ein Simulationskern lediglich einmal implementiert werden und kann anschließend bei jeder Erstellung neuer Simulationsprogramme unverändert wiederverwendet werden. Insbesondere auf dieser Erkenntnis basierend, entstand das Konzept der im folgenden Abschnitt vorgestellten Simulationssysteme.

Ein Experiment am Simulationsmodell erfolgt in der Computersimulation durch die Ausführung eines *Simulators*. Eine derartige Ausführung wird als *Simulationslauf* (*simulation run*) bezeichnet. Bei dem dabei ausgeführten Simulator handelt es sich um die Gesamtheit aller Komponenten, die für die Ausführung des Simulationsprogrammes benötigt werden: Hardware, Betriebssystem und das Simulations-

programm selbst. Eine schematische Darstellung des beschriebenen Aufbaus ist in Abbildung 2.1 zu sehen.

Die initialen Variablenbelegungen können zwar prinzipiell auch innerhalb des Simulationsmodells durch die Angabe von konstanten Initialwerten definiert werden. Es ist allerdings wesentlich praktischer, das Simulationsmodell parametrisiert zu gestalten, d. h. die initiale Belegung der Zustandsvariablen abhängig von der Belegung sogenannter *Simulationsparameter* zu variieren. Diese Simulationsparameter werden erst zu Beginn eines Simulationslaufes gesetzt, so dass es möglich ist, einmal fertig erstellte Simulationsprogramme nicht nur für ein Experiment, sondern für ganze Experimentserien verwenden zu können.

Neben den Simulationsparametern existiert noch eine weitere Klasse von Parametern: die sogenannten *Laufzeitparameter*. Dabei handelt es sich um Parameter für den Simulationskern, die das Verhalten des Simulationsmodells nicht verändern und damit auch die Entwicklung des Modellzustandes nicht beeinflussen. Die einzige potentielle Auswirkung von Laufzeitparametern besteht in der Veränderung der benötigten Zeitdauer, die für eine Ausführung des Simulationsmodells benötigt wird.

Simulationssystem

Ein *Simulationssystem* ist eine Werkzeugsammlung zur Erstellung eigener Simulationsprogramme. Nach [Fis82] besteht ein Simulationssystem u. a. aus folgenden Komponenten:

- der Definition einer Simulationssprache, mittels der Simulationsmodelle formal als Programm beschrieben werden können,
- einem Compiler, der derartig notierte Simulationsmodelle in ausführbare Programmteile (Objektcode) übersetzt bzw. einem Interpreter, der die Simulationsmodelle direkt ausführen kann,
- einem Laufzeitsystem, das während eines Simulationslaufes die Steuerung und Beobachtung des Simulationsmodells sowie simulationsferne Aufgaben (Speicherallokation, Ein-/Ausgabe, ...) übernimmt (der bereits erwähnte Simulationskern) und
- bereits vorimplementierten Standardalgorithmen zur Vereinfachung der Simulationsmodellentwicklung (Zufallszahlengeneratoren, statistische Hilfsfunktionen, ...).

Mit Hilfe eines Simulationssystems lassen sich prinzipiell Simulationsprogramme für beliebige Simulationsmodelle entwickeln, solange sich diese Modelle in der vom Simulationssystem vorgegebenen Simulationssprache formulieren lassen. Weitere Einschränkungen können sich jedoch aus der Implementation des Simulationskerns sowie der eingesetzten Rechentechnik ergeben.

Von einer *Simulationsbibliothek* (*simulation library*) spricht man, wenn der Spezialfall eines Simulationssystems vorliegt, bei dem

- die Simulationssprache mit einer existierenden Programmiersprache identisch ist (zusätzliche, programmiersprachenkonforme Definitionen von Klassen, Methoden, u. ä. ausgenommen),
- das Laufzeitsystem und die vorimplementierten Hilfsalgorithmen ebenfalls in dieser Programmiersprache vorliegen und
- die selbst erstellten Simulationsprogramme demzufolge direkt mit einem Standardcompiler der Programmiersprache generiert werden können, also das Simulationssystem keinen separaten Compiler erfordert.

In der vorliegenden Arbeit wird der Schwerpunkt bei Simulationssystemen in Form von Simulationsbibliotheken liegen.

2.3 Zeitdefinitionen

Im Kontext von Simulationen existieren verschiedene Zeitbegriffe, die sorgfältig unterschieden werden müssen. Allerdings gibt es bis heute kein einheitliches Benennungsschema der verschiedenen Zeitbegriffe in der Simulationsliteratur. Im Folgenden werden die in der vorliegenden Arbeit verwendeten Zeitbegriffe eingeführt, die weitestgehend mit denen aus den Werken von Schwarze und Fischer [Sch90, Fis82] übereinstimmen. Die in Klammern genannten englischen Bezeichnungen beziehen sich auf das Standardwerk zur parallelen Simulation von Fujimoto [Fuj99].

Die *Realzeit* (*physical time*) entspricht der „realen“ Zeit, die im zu analysierenden, originalen System vergeht. Im Falle der Simulation eines Eisenbahnnetzes würde also ein Zeitpunkt der Realzeit aus den üblichen Datums- und Uhrzeitkomponenten wie Jahr, Monat, Tag, Stunde, Minute und Sekunde bestehen.

Die abstrakte Zeit, die während eines Simulationslaufes im Simulationsmodell vergeht, also die Repräsentation der Realzeit im Modell darstellt, bezeichnet man als *Modellzeit* (*simulation time*). Diese wird oft zur Vereinfachung der Simulationsimplementation durch einen einfachen Zahlentyp dargestellt. Damit die Ergebnisse der Simulation auf das ursprüngliche System übertragen werden können, muss eine Relation zwischen der Modellzeit und der Realzeit definiert sein. Üblicherweise handelt es sich dabei um eine lineare Abhängigkeit zwischen beiden Zeiten, so dass auch Zeitintervalle aus einer Zeit direkt in die andere übertragen werden können.

Die „reale“ Zeit, die während eines Simulationslaufes vergeht, wird als *Ausführungszeit* (*wallclock time*)[1] bezeichnet. Sie ist die Grundlage für alle Messungen und quantitativen Aussagen bezüglich der Geschwindigkeit von Simulationsläufen.

[1] Die englische Bezeichnung basiert auf der Vorstellung, dass ein Simulierender die Ausführungszeit jederzeit an einer Wanduhr des Labors ablesen könnte.

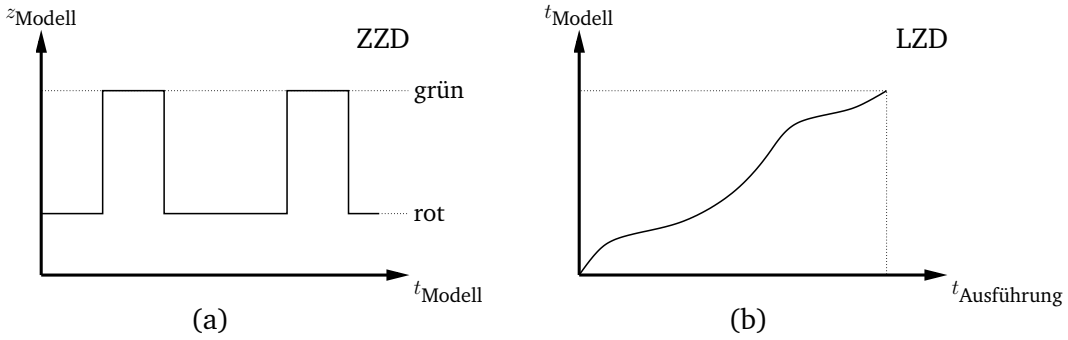

Abbildung 2.2: Beispiel eines Zustand-Zeit- und eines Laufzeitdiagrammes

2.4 Graphische Darstellung von Simulationsläufen

Zustand-Zeit-Diagramme

Eine in der Simulationsliteratur häufig anzutreffende graphische Darstellung von Simulationsläufen sind sogenannte Zustand-Zeit-Diagramme. Bei dieser Darstellung werden in einem kartesischen Koordinatensystem auf der Abszisse die Modellzeit t_{Modell} und auf der Ordinate eine eindimensionale Projektion des Zustandes des Simulationsmodells z_{Modell} dargestellt.

In Abbildung 2.2a ist als Beispiel das Zustand-Zeit-Diagramm eines Simulationslaufes einer einfachen Ampelsimulation zu sehen. In dieser Arbeit sind alle Zustand-Zeit-Diagramme in der rechten oberen Ecke mit dem Kürzel ZZD versehen.

Laufzeitdiagramme

Zusätzlich zu den Zustand-Zeit-Diagrammen werden im weiteren Verlauf der Arbeit an mehreren Stellen Laufzeitdiagramme zur Verdeutlichung des Zusammenhangs zwischen Ausführungszeit und Modellzeit während eines Simulationslaufes verwendet. Bei dieser Darstellung des Zeit(en)verlaufes eines Simulationslaufes werden in einem kartesischen Koordinatensystem auf der Abszisse die Ausführungszeit $t_{\text{Ausführung}}$ und auf der Ordinate die zugehörige Modellzeit t_{Modell} abgetragen.[2]

In Abbildung 2.2b ist exemplarisch das Laufzeitdiagramm eines einfachen Simulationslaufes dargestellt. Insbesondere zur schnelleren und leichteren Unterscheidung von den Zustand-Zeit-Diagrammen besitzen alle Laufzeitdiagramme in dieser Arbeit in der rechten oberen Ecke das Kürzel LZD.

Die gestrichelten Linien geben die Ausführungs- und Modellzeitpunkte an, bei denen der Simulationslauf endet. Daraus folgt, dass die eingetragene Kurve stets im Schnittpunkt dieser beiden gestrichelten Linien enden muss.

[2] Statt der Modellzeit kann alternativ auch die Realzeit abgetragen werden. Allerdings wird sich dabei die Darstellung in den meisten Simulationen nicht qualitativ ändern, da in der Regel ein linearer Zusammenhang zwischen Modell- und Realzeit besteht.

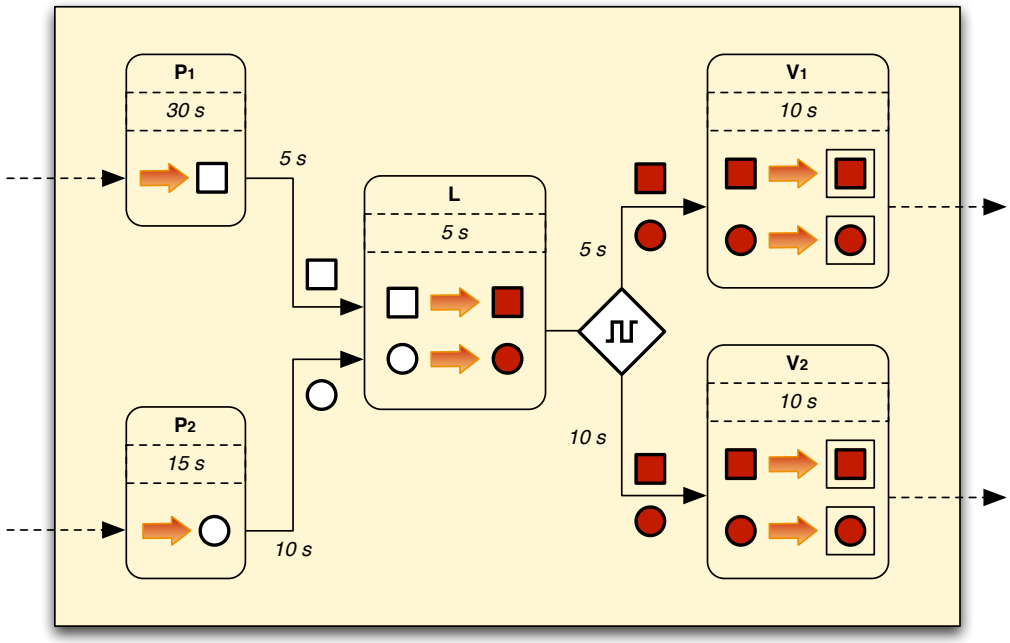

Abbildung 2.3: Überblick über das Beispielszenario

2.5 Beispielszenario

Zum besseren Verständnis der bereits vorgestellten Grundbegriffe sowie der später noch beschriebenen Klassifikationen und Simulationsalgorithmen soll das folgende Beispielszenario dienen.

Gegeben sei eine Spielzeugfabrik, die bunte Bauklötze in den zwei Formvarianten Zylinder und Würfel herstellt. Diese Fabrik, deren Aufbau schematisch in Abbildung 2.3 zu sehen ist, besteht aus folgenden Elementen:

- zwei Produktionsstationen, die aus Rohmaterial Bauklötze herstellen, wobei die erste Produktionsstation (P_1) alle 30 Sekunden einen Würfel erzeugt, während die zweite (P_2) alle 15 Sekunden einen Zylinder produziert,
- einer Lackierstation (L), die für das Lackieren eines Bauklotzes 5 Sekunden benötigt,
- zwei Verpackungsstationen (V_1 und V_2), die abwechselnd von der Lackierstation beliefert werden und jeweils 10 Sekunden für das Verpacken eines Bauklotzes benötigen, sowie
- Transportbändern zwischen den einzelnen Stationen (in Abbildung 2.3 als Pfeile zwischen den Stationen dargestellt), deren Transportzeiten bei 5 bzw. 10 Sekunden liegen.

Im Initialzustand befinden sind alle Stationen und Transportbänder frei von Bauklötzen. Diese entstehen ausschließlich durch die Arbeit der beiden, gleichzeitig zu Beginn des Szenarios startenden, Produktionsstationen.

Das beschriebene Szenario lässt sich leicht als System auffassen: Der umschließende Kasten in der Abbildung markiert die Systemgrenze; die gestrichelten Pfeile stehen für die Ein- und Ausgaben des Systems in Form von Rohmaterial bzw. fertig verpackten Bauklötzen. Der Einfachheit halber wird in den folgenden Abschnitten, in denen verschiedene Simulationsmodelle dieses Systems beschrieben werden, von den Ein- und Ausgaben abstrahiert. Des Weiteren wurden zur Vereinfachung die verschiedenen Zeiten im Beispielszenario bewusst so gewählt, dass es niemals zu einer Stauung vor einer Station kommen kann, d. h., jede Lackier- oder Verpackungsstation befindet sich bei jedem Eintreffen eines Bauklotzes im Leerlauf.

2.6 Klassifikation von Simulationen

Simulationen lassen sich nach verschiedenen Kriterien klassifizieren, wobei diese Klassifikationen auf Basis der Eigenschaften der zugehörigen Simulationsmodelle oder der eingesetzten Simulationssysteme vorgenommen werden. Im Folgenden werden die für diese Arbeit relevanten Klassifikationen vorgestellt.

Klassifikation nach Zeitentwicklung

Eine erste simulationssystembezogene Klassifikation basiert auf dem Zusammenhang zwischen den drei verschiedenen Zeitkonzepten, die in Abschnitt 2.3 vorgestellt wurden.

Von *skalierten Echtzeit-Simulationen* spricht man, wenn während eines Simulationslaufes die der Modellzeit entsprechende Realzeit proportional zur Ausführungszeit voranschreitet. Einen Spezialfall der skalierten Echtzeit-Simulationen bilden die *Echtzeit-Simulationen*: hier wachsen Real- und Ausführungszeit synchron. Skalierte Echtzeit-Simulationen werden üblicherweise benutzt, wenn während eines Simulationslaufes eine Animation stattfindet, die dem Simulierenden die auftretenden Änderungen im Simulationsmodell verdeutlichen soll.

In der vorliegenden Arbeit werden hingegen ausschließlich sogenannte *As-fast-as-possible-Simulationen* betrachtet. Bei dieser Klasse von Simulationen wird überhaupt kein konkreter Zusammenhang zwischen Modell- und Ausführungszeit angestrebt. Stattdessen wird versucht, das Simulationsmodell so schnell wie möglich auszuführen.

Der Unterschied zwischen einer (skalierten) Echtzeit- und einer As-fast-as-possible-Simulation ist im Laufzeitdiagramm gut zu erkennen (siehe Abbildung 2.4). Während im Laufzeitdiagramm der skalierten Echtzeit-Simulation der erwartete lineare Zusammenhang zwischen Modell- und Ausführungszeit deutlich sichtbar ist, schreitet die Modellzeit bei einer As-fast-as-possible-Simulation mal schneller und mal langsamer im Vergleich zur Ausführungszeit voran. Selbst die Extremfälle eines

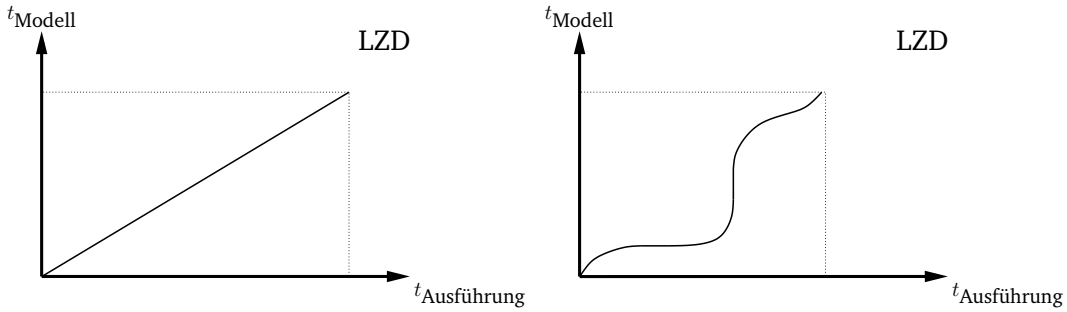

Abbildung 2.4: Gegenüberstellung der Laufzeitdiagramme einer skalierten Echtzeit- und einer As-fast-as-possible-Simulation

temporären Stagnierens in einer der beiden Zeiten (erkennbar an einem horizontalen oder vertikalen Abschnitt im Graphen) sind möglich.

Ein Verharren in der Ausführungszeit bei gleichzeitigem Fortschreiten der Modellzeit (vertikaler Abschnitt im Laufzeitdiagramm) tritt immer dann auf, wenn während des Simulationslaufes der Zustand des Simulationsmodells für eine Zeitspanne (in der Modellzeit) konstant bleibt. Eine As-fast-as-possible-Simulation „springt“ in einer solchen Situation zur nächsten Zustandsänderung, so dass im Laufzeitdiagramm ein vertikaler Abschnitt entsteht. Ein Stagnieren in der Modellzeit bei gleichzeitigem Fortschreiten der Ausführungszeit (horizontaler Abschnitt im Laufzeitdiagramm) entsteht hingegen immer dann, wenn während des Simulationslaufes die Zustandsberechnung des Simulationsmodells soviel (Ausführungs-)Zeit benötigt, dass dadurch das Fortschreiten in der Modellzeit verzögert wird.

Zeitkontinuierliche vs. zeitdiskrete Simulation

Eine Klassifikation von Simulationen anhand der verwendeten Simulationsmodelle ist die Unterteilung in zeitkontinuierliche und zeitdiskrete Simulationen.

Als *zeitkontinuierlich* (oft auch nur kurz: *kontinuierlich*) bezeichnet man Simulationen, bei deren Simulationsmodellen sich der Modellzustand kontinuierlich mit der Modellzeit ändern kann. Meistens werden derartige Simulationsmodelle durch Algebro-Differentialgleichungssysteme beschrieben, die während eines Simulationslaufes numerisch gelöst werden.[3]

Von *zeitdiskreten* (analog auch oft nur kurz: *diskreten*) Simulationen spricht man hingegen, wenn das Simulationsmodell in jedem endlichen Modellzeitintervall nur eine endliche Anzahl von Zustandsänderungen zulässt. Dabei muss betont werden, dass auch in diskreten Simulationen sowohl die Zustände als auch die Modellzeit kontinuierliche Werte annehmen können.

[3] Obwohl während eines solchen Simulationslaufes auf einem Digitalrechner zwangsläufig eine Diskretisierung einhergeht, ändert sich nichts an der Klassifikation als kontinuierliche Simulation. Insbesondere besteht (wenn auch heutzutage eher theoretisch) durchaus die Möglichkeit, ein kontinuierliches Simulationsmodell ohne Diskretisierung auf einem Analogrechner auszuführen.

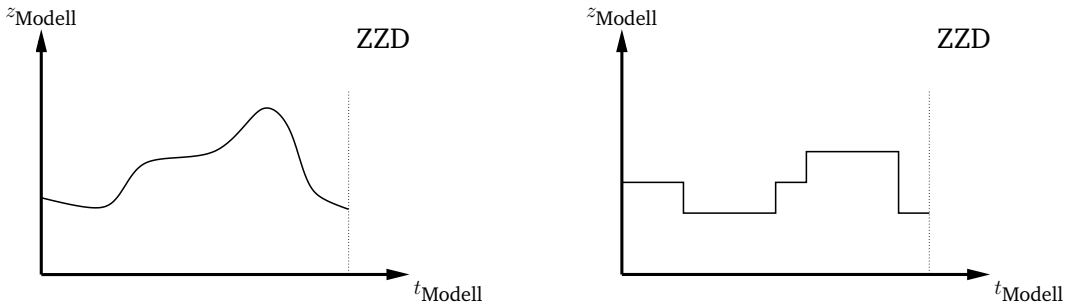

Abbildung 2.5: Gegenüberstellung der Zustand-Zeit-Diagramme eines kontinuierlichen und eines diskreten Simulationslaufes

In Abbildung 2.5 sind die Zustand-Zeit-Diagramme je eines kontinuierlichen und eines diskreten Simulationslaufes zu sehen.

Zeitgesteuerte vs. ereignisgesteuerte Simulation

Diskrete Simulationen lassen sich, basierend auf der Arbeitsweise des verwendeten Simulationssystems, weiter unterteilen in *zeitgesteuerte* (*time-stepped*) und *ereignisgesteuerte* (*event-driven*) Simulationen.

Bei zeitgesteuerten Simulationen erfolgt ein Simulationslauf durch ein Berechnen/Betrachten des Simulationsmodellzustandes zu vorher festgelegten, in der Regel äquidistanten Modellzeitpunkten. Dabei spielt es keine Rolle, ob zu diesen Zeitpunkten wirklich eine Zustandsänderung eintritt oder auch nur eintreten kann. Demzufolge kann eine zeitgesteuerte Simulation je nach Simulationsmodell einen Großteil der Simulationslaufzeit mit der unnötigen Betrachtung für das Simulationsergebnis irrelevanter Zeitpunkte verbringen.

Ein Anwendungsfall der zeitgesteuerten Simulation besteht in der Realisierung von skalierten Echtzeitsimulationen mit angeschlossener Animation. Wenn die Berechnung der Zustandsänderungen im Simulationsmodell im Vergleich zur Gesamtlaufzeit eines Simulationslaufes vernachlässigbar kurz dauert, ist die zeitgesteuerte Simulation ein probates Mittel, um einen bei Animationen gewünschten konkreten (in der Regel linearen) Zusammenhang zwischen Modellzeit und Ausführungszeit herzustellen.

Bei der ereignisgesteuerten Simulation werden hingegen ausschließlich die Modellzeitpunkte betrachtet, bei denen eine Änderung des Simulationsmodellzustandes zumindest potentiell eintritt. Eine solche Zustandsänderung bezeichnet man als *Ereignis* (*event*); der zugehörige Zeitpunkt in der Modellzeit wird auch als *Zeitstempel* des Ereignisses bezeichnet.

In Abbildung 2.6 sind die Zustand-Zeit-Diagramme eines zeitgesteuerten und eines ereignisgesteuerten Simulationslaufes basierend auf demselben Simulationsmodell zu sehen. Die senkrechten, dünnen Linien sollen dabei die Berechnungsschritte markieren und den Mehraufwand bei der zeitgesteuerten Simulation verdeutlichen.

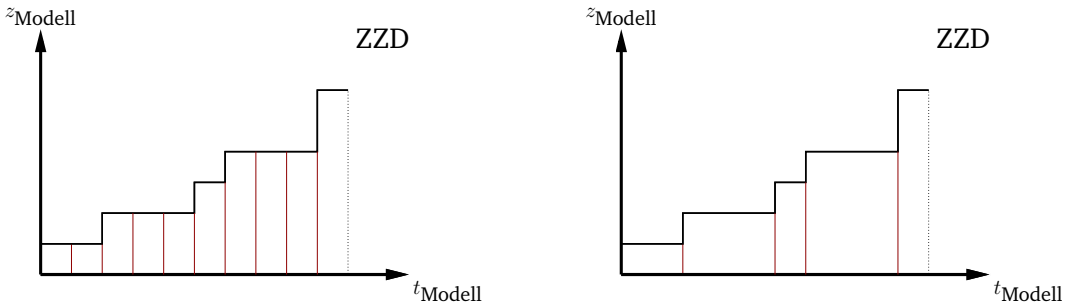

Abbildung 2.6: Gegenüberstellung der Zustand-Zeit-Diagramme eines zeitgesteuerten und eines ereignisgesteuerten Simulationslaufes

Erwähnenswert ist die Tatsache, dass eine zeitgesteuerte Simulation durch eine ereignisgesteuerte Simulation „simuliert“ werden kann. Dazu müssen lediglich letzterer zusätzliche Ereignisse, sogenannte Zeitereignisse, hinzugefügt werden, die zu äquidistanten Modellzeitpunkten eintreten. Dabei muss das Modellzeitintervall zwischen diesen Zeitereignissen so gewählt werden, dass jeder Eintrittszeitpunkt eines ursprünglichen „echten“ Ereignisses mit dem Eintrittszeitpunkt eines Zeitereignisses zusammenfällt. Dieser Ansatz wird vor allem gewählt, um ereignisgesteuerte Simulationen nachträglich mit einer Animation zu versehen. Im Idealfall erfolgt das Hinzufügen von Zeitereignissen in einer Form, die ein später frei wählbares An- und Abschalten der Zeitereignisse erlaubt. Dadurch können spätere Simulationsläufe wahlweise ereignisgesteuert in hoher Geschwindigkeit (da ohne Zeitereignisse) ohne Animation oder aber zeitgesteuert bei verminderter Geschwindigkeit mit Animation durchgeführt werden [Eve06, Hel07].

2.7 Reproduzierbarkeit von Simulationen

Eine zentrale Anforderung an jede Simulationsimplementation ist die *Reproduzierbarkeit* der Simulationsläufe. Damit ist die Forderung gemeint, dass jede wiederholte Ausführung desselben Simulationsprogrammes bei gleichen Umgebungsbedingungen (gleicher Initialzustand des Simulationsmodells, gleiche Eingaben bei interaktiven Simulationen, . . .) identische Simulationsergebnisse liefert.

Ein Grund für die Forderung nach Reproduzierbarkeit liegt in der deutlich erleichterten Fehlersuche bei der Entwicklung und Wartung von Simulationsmodellen. In Simulationen, die nichtreproduzierbare Ergebnisse liefern, sind die Fehler häufig ebenfalls nichtreproduzierbar und demzufolge deutlich schwerer zu lokalisieren. Des Weiteren kann der Erfolg einer Fehlerkorrektur in nichtreproduzierbaren Simulationen mitnichten durch einen Vergleich mit dem Simulationsergebnis eines weiteren Simulationslaufes überprüft werden. Darüber hinaus erhöht sich die Glaubwürdigkeit von Simulationsergebnissen, wenn diese durch Wiederholungen reproduzierbarer Simulationen bestätigt werden können.

Nichtdeterministische Simulationsmodelle

Die Hauptursache für nichtreproduzierbare Simulationen liegt in der Verwendung von nichtdeterministischen Simulationsmodellen, d. h. Simulationsmodellen, deren Verhalten nicht vollständig deterministisch spezifiziert wurde. In der Praxis sind damit stochastische Simulationsmodelle gemeint, bei denen das Simulationsmodell explizit vorgibt, dass während einer Simulation Teile des Simulationsmodellzustandes und/oder Eintrittszeitpunkte von Ereignissen zufällig generiert werden. Da eine grundlegende Eigenschaft von zufälligen Werten darin besteht, gerade nicht reproduzierbar zu sein, sind nichtdeterministische Simulationsmodelle mit der Reproduzierbarkeit eines Simulationslaufes unvereinbar.

In diesem Fall besteht ein üblicher Ausweg in der Verwendung von Pseudozufallszahlengeneratoren, die, abhängig von einem Startwert, reproduzierbare Folgen von Pseudozufallszahlen generieren. Verwendet man in einem nichtdeterministischen Simulationsmodell ausschließlich derartige Pseudozufallszahlengeneratoren und zählt die Startwerte für selbige zum Initialzustand des Simulationsmodells, so überführt man dadurch das nichtdeterministische Simulationsmodell in ein deterministisches, wodurch die Reproduzierbarkeit hergestellt wird.

Gleichzeitig eintretende Ereignisse

Bei ereignisgesteuerten Simulationen wird das gesamte Verhalten eines Simulationsprogrammes ausschließlich durch die auftretenden Ereignisse bestimmt. Bei der Verwendung von deterministischen Simulationsmodellen sind die mit diesen Ereignissen verbundenen Auswirkungen, insbesondere die Generierung neuer Ereignisse, bedingt durch die deterministischen Verhaltensbeschreibungen ebenfalls identisch.

Unter diesen Voraussetzungen besteht die einzige Möglichkeit der Nichtreproduzierbarkeit eines Simulationsprogrammes in einer veränderten Reihenfolge der Abarbeitung der Ereignisse. Daraus ergibt sich auch die Definition der Reproduzierbarkeit nach Mehl [Meh94], die stillschweigend ein deterministisches Verhalten von Ereignissen bzgl. des Simulationsmodellzustandes voraussetzt:

> „Eine ereignisgesteuerte Simulation wird *reproduzierbar* genannt, wenn aus Sicht eines externen Beobachters bei jeder Wiederholung desselben Simulationsexperiments alle Ereignisse in der gleichen Reihenfolge ausgeführt werden.“

In sequentiellen deterministischen Simulationsprogrammen gibt es genau einen Grund, warum Ereignisse in einem erneuten Simulationslauf in einer anderen Reihenfolge abgearbeitet werden können: die Existenz von Ereignissen mit demselben Eintrittszeitpunkt (bzgl. Modellzeit). Dieser Fall kann durch eine Priorisierung der beteiligten Ereignisse gelöst werden. Dabei gibt es wahlweise die Möglichkeit der expliziten und der impliziten Priorisierung. Bei einer expliziten Priorisierung ist der Simulationsmodellentwickler dafür verantwortlich, für jedes Ereignis bzw. für jeden

Ereignistyp eine Priorität festzulegen, anhand derer beim gleichzeitigen Auftreten von Ereignissen die Reihenfolge der Abarbeitung entschieden wird.

Bei einer impliziten Priorisierung legt hingegen der Simulationskern selbst fest, in welcher Reihenfolge gleichzeitig auftretende Ereignisse bearbeitet werden. Ein häufig anzutreffender, kanonischer Ansatz besteht darin, gleichzeitige Ereignisse in der Reihenfolge der Eintrittszeitpunkte ihrer Erzeugerereignisse abzuarbeiten. Eine Verfälschung des Simulationsergebnisses durch eine implizite Priorisierung ist dabei nicht zu befürchten, da gleichzeitige Ereignisse prinzipiell nur dann auftreten können, wenn das Simulationsmodell diese explizit zulässt und in diesem Fall jede beliebige Abarbeitungsreihenfolge gültig ist.

2.8 Implementation ereignisgesteuerter Simulationen

Wie bereits in Abschnitt 2.2 erwähnt, besteht ein Simulationsprogramm prinzipiell aus zwei logischen Komponenten, die auch separat implementiert werden sollten: dem Simulationsmodell und dem Simulationskern. Die Grundidee hinter dieser Trennung besteht darin, den gleichen, modellunabhängigen Simulationskern in einer Vielzahl von Simulationsprogrammen wiederverwenden zu können.

Wie ebenfalls bereits beschrieben wurde, besteht ein Simulationsmodell aus zwei Bestandteilen: Zustandsvariablen, deren Belegungen in ihrer Gesamtheit den Zustand des originalen Systems repräsentieren und Verhaltensmethoden, die das dynamische Verhalten des Systems nachbilden. Wie genau die Zustandsvariablen und Verhaltensmethoden im Simulationsmodell umgesetzt werden, hängt dabei maßgeblich davon ab, welche der im Folgenden diskutierten *Modellierungssichten* bei der Modellerstellung verwendet wird.

In der Praxis haben sich dabei zwei Modellierungssichten durchgesetzt: die *ereignisorientierte* (*event-oriented*) und die *prozessorientierte* (*process-oriented*) Sicht. In der Simulationsliteratur findet man bisweilen die *aktivitätsorientierte* und deren Spezialfall, die *transaktionsorientierte* Sicht als weitere Modellierungssichten. Da sich allerdings Simulationsmodelle, die auf einer dieser beiden Sichten basieren, prinzipiell nicht effizient implementieren lassen, eignen sich diese fast ausschließlich für theoretische Betrachtungen [Hel00].

Ereignisorientierte Simulationsmodelle

Bei einer ereignisorientierten Modellierung wird das Simulationsmodell des zu analysierenden Systems aus einer globalen Sicht erstellt. Es werden systemweit alle potentiell auftretenden Ereignistypen identifiziert und zu jedem wird im Rahmen einer sogenannten *Ereignisroutine* beschrieben, wie ein konkretes Ereignis dieses Typs den Zustand des Modells verändert und welche zukünftigen Ereignisse das konkrete Ereignis zur Folge hat. In objektorientierten Sprachen werden Ereignistypen üblicherweise als Klassen realisiert, von denen während eines Simulationslaufes die auftretenden Ereignisse instantiiert werden.

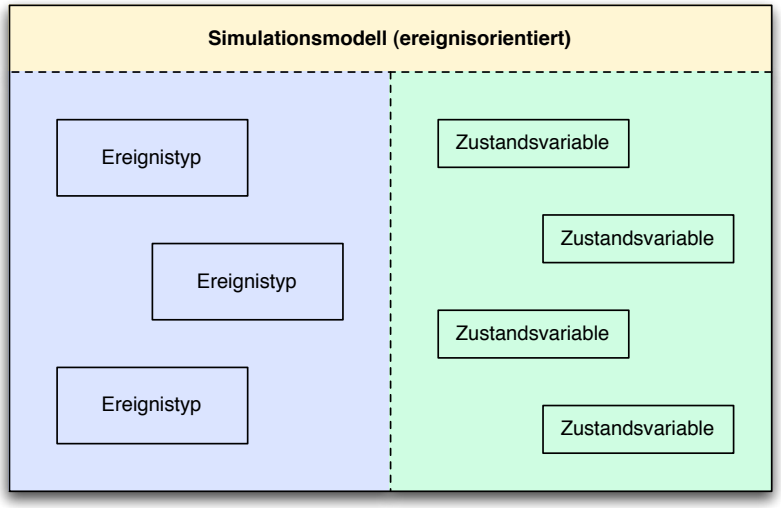

Abbildung 2.7: Aufbau eines ereignisorientierten Simulationsmodells

```
1 void eventroutine() {
2     create new event at L at (current_time + 5)
3     create new event at P1 at (current_time + 30)
4 }
```

Quelltext 2.1: Ereignisroutine der Ankunftsereignisse an einer Produktionsstation

Damit Ereignisse den Simulationsmodellzustand ändern können, müssen die Zustandsvariablen von den Ereignisroutinen aus erreichbar sein. In der Regel wird dies dadurch erreicht, dass die Zustandsvariablen auf einem globalen Niveau bzw. in objektorientierten Sprachen als öffentliche Variablen global erreichbarer Simulationsmodellobjekte deklariert werden und damit allen Ereignisroutinen gleichermaßen zugänglich sind.

In Abbildung 2.7 ist der schematische Aufbau eines ereignisorientierten Simulationsmodells, bestehend aus Ereignistypen und Zustandsvariablen, zu sehen.

Wenn man das in Abschnitt 2.5 beschriebene Beispielsystem der Spielzeugfabrik ereignisorientiert modelliert, so könnte ein erster Ereignistyp in der Produktion eines Würfels in der ersten Produktionsstation bestehen. Jedes Ereignis dieses Ereignistyps bedingt zwei Folgeereignisse: ein Ankunftsereignis des Würfels an der Lackierstation nach fünf Sekunden (die Zeit für den Transport zwischen beiden Stationen) sowie ein weiteres Würfelproduktionsereignis nach 30 Sekunden (das Intervall zwischen zwei Würfelproduktionen). In Pseudocode würde die Ereignisroutine des Produktionsereignistyps der ersten Produktionsstation (P_1) demnach wie in Quelltext 2.1 aussehen.

Die Ereignisroutine des Ankunftsereignistyps der Lackierstation (L), abgebildet in Quelltext 2.2, ist etwas aufwendiger, da hier zusätzlich das abwechselnde Beschi-

```
1 boolean oben = true; // globale (Zustands−)Variable
2 void eventroutine() {
3     if (oben) create new event at V1 at (current_time + 5 + 5)
4     else create new event at V2 at (current_time + 5 + 10)
5     oben = !oben
6 }
```

Quelltext 2.2: Ereignisroutine der Ankunftsereignisse an der Lackierstation

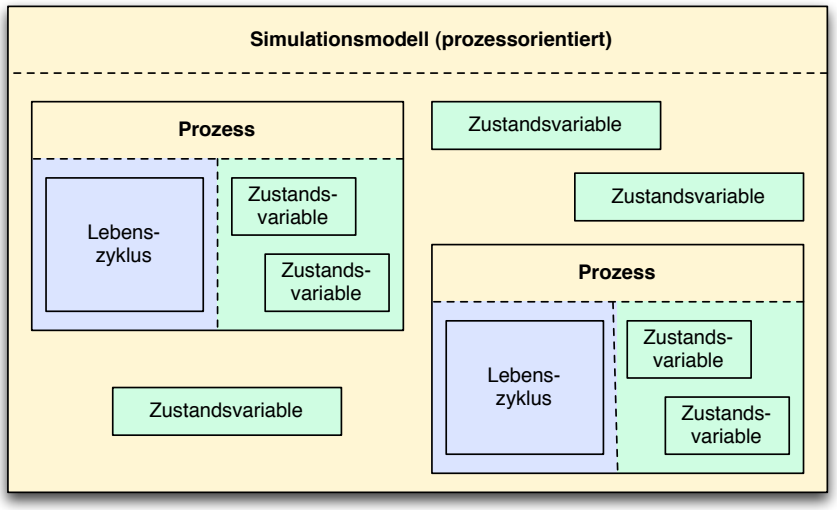

Abbildung 2.8: Aufbau eines prozessorientierten Simulationsmodells

cken der Verpackungsstationen berücksichtigt werden muss. Dabei ist zu beachten, dass sich die dafür benötigte Variable nicht im Kontext des Ereignisses befinden darf, da dessen Existenz ja nach seiner Abarbeitung beendet wird. Folglich wird eine Variable auf globalem Niveau (im Beispiel die Variable *oben*) benötigt, die das Wissen, welche Verpackungsstation als letzte beliefert wurde, für das nächste Ereignis der Lackierstation speichert.

Die Ereignisroutinen für die Produktion an der zweiten Produktionsstation und die Ankünfte an den beiden Verpackungsstationen sind analog zu erstellen.

Prozessorientierte Simulationsmodelle

In einer prozessorientierten Modellierung wird das zu analysierende System als eine Menge interagierender *Prozesse* gesehen (Abbildung 2.8). Ein Prozess stellt dabei einen Systemteil dar, der einen eigenen Zustand (als Teil des Systemzustands) besitzt, eine wohldefinierte Teilfunktionalität des Systems erbringt und nur begrenzt mit dem restlichen System interagiert. Er kann also nach den Definitionen von Gaines und Ashby selbst wieder als ein System aufgefasst werden (in der Systemde-

finition von Karplus wird potentiell der Aspekt der Kausalität – die Eingaben des Systems dürfen nie von seinen Ausgaben abhängig sein – verletzt).

Zu jedem Prozesstyp formuliert man eine *Prozessroutine*, die jedoch im Unterschied zur Ereignisroutine nicht nur die Zustandsänderungen und Simulationsfortführung zu einem bestimmten Modellzeitpunkt, sondern über die gesamte Lebenszeit eines Prozesses dieses Typs, mitunter über den gesamten betrachteten Modellzeitraum, beschreibt. Dieses wird durch zwei Hilfsfunktionen ermöglicht: das Unterbrechen eines Prozesses sowie das Fortsetzen desselbigen zu einem späteren Zeitpunkt.[4] Dabei kann die Unterbrechung eines Prozesses ausschließlich durch diesen selbst ausgelöst werden, während ein Prozess sowohl bei anderen Prozessen als auch bei sich selbst eine Prozessfortsetzung veranlassen kann (letzteres natürlich nur zu einem vorbestimmten Zeitpunkt in der Zukunft bzgl. der Modellzeit).

Oft besitzt eine Prozessroutine einen zyklischen Aufbau, weshalb sie auch *Lebenszyklus* (*life cycle*) des Prozesstyps bzw. des Prozesses genannt wird. In objektorientierten Sprachen bietet es sich an, Prozesstypen als Klassen zu implementieren. In diesem Fall werden die prozessinternen Zustandsvariablen als private Attribute und der Lebenszyklus als Methode realisiert.

Bei der prozessorientierten Modellierung sind auch Zustandsvariablen außerhalb der Prozesse erlaubt. Diese auch als *Ressourcen* bezeichneten Variablen werden zur Repräsentation von denjenigen Teilen des Systemzustandes verwendet, die durch mehrere Prozesse beeinflusst werden können.

Im Beispielszenario besteht eine Modellierungsmöglichkeit in der Betrachtung der einzelnen Stationen als Prozesstypen, von denen während eines Simulationslaufes jeweils genau ein Prozess gebildet wird. Dabei besitzen die beiden Prozesstypen für die Produktionsstationen einen einfachen Lebenszyklus: In jeder Iteration einer Endlosschleife sorgen sie zuerst für eine Fortsetzung des Prozesses der Lackierstation nach 5 bzw. 10 Sekunden und warten anschließend für 30 bzw. 15 Sekunden. In Quelltext 2.3 ist die Umsetzung dieses Prozesslebenszyklus' in Pseudocode dargestellt.

Der Prozesstyp der Lackierstation ist etwas komplizierter aufgebaut. Hervorzuheben ist die Tatsache, dass im Unterschied zum ereignisorientierten Modell hier keine globale Variable benötigt wird, um die Information über die nächste zu beliefernde Verpackungsstation zu speichern. Stattdessen befindet sich diese Information im konkreten Prozess selbst, wie in Quelltext 2.4 zu sehen ist.

Ob ein Simulationsmodell ereignisorientiert oder prozessorientiert erstellt werden sollte, ist stark vom zu simulierenden System abhängig. Generell lässt sich festhal-

[4] Für das Zusammenspiel von Unterbrechen und Fortsetzen von Prozessen gibt es in Simulationsliteratur und -systemen mannigfaltige Bezeichnungen: Übliche Synonyme für das *Unterbrechen* (*suspend*) sind *Warten* (*wait*), *Schlafen* (*sleep*) und *Deaktivieren* (*deactivate*); anstatt des *Fortsetzens* (*continue* oder auch *resume*) findet man bisweilen auch *Aufwecken* (*wake up*) und *Aktivieren* (*activate*). Insbesondere *Warten* und *Schlafen* sind problematisch, da einige Autoren bzw. Simulationssysteme darunter nicht das potentiell dauerhafte Unterbrechen eines Prozesses, sondern das Unterbrechen für eine vordefinierte Zeitdauer inklusive seiner anschließenden Fortsetzung verstehen.

```
1 void lifecycle () {
2     while (true) {
3         activate process L at (current_time + 5)
4         activate myself at (current_time + 30)
5         suspend()
6     }
7 }
```

Quelltext 2.3: Prozesslebenszyklus einer Produktionsstation

```
1  void lifecycle () {
2      boolean oben = true;
3      suspend() // warten auf den ersten Bauklotz
4      while (true) {
5          sleep(5) // Zeit des Lackierens
6          if (oben) activate process V1 at (current_time + 5)
7          else activate process V2 at (current_time + 10)
8          oben = !oben
9          suspend() // warten auf den naechsten Bauklotz
10     }
11 }
```

Quelltext 2.4: Prozesslebenszyklus der Lackierstation

ten, dass die prozessorientierte Sicht in der Regel zu übersichtlicheren Modellen führt. Dies liegt vor allem darin begründet, dass sich eine bereits im Originalsystem befindliche Gliederung in Teilsysteme in den Prozessen des Simulationsmodells wiederfindet, d. h., prozessorientierte Modelle weisen in der Regel ein hohes Maß an Strukturäquivalenz mit dem Originalsystem auf. Insbesondere durch die Systemeigenschaft von Prozessen gliedert sich die prozessorientierte Modellierung auch gut als Arbeitsschritt in die in Abschnitt 2.2 beschriebene, iterative System-/Modellbildung ein.

Der größte Nachteil der prozessorientierten gegenüber der ereignisorientierten Modellierung besteht in einer höheren benötigten Gesamtlaufzeit der zugehörigen Simulationen. Wie noch detailliert in Kapitel 4 beschrieben wird, ist dies darauf zurückzuführen, dass jedwede Realisierung von Prozessen immer mit einem Mehraufwand zur Simulationslaufzeit verbunden ist. Des Weiteren lassen sich Prozesse in zahlreichen aktuellen Programmiersprachen nicht direkt implementieren, wie in Kapitel 4 noch detailliert diskutiert wird. Zumindest dieser Nachteil ist jedoch für einen Simulationsmodellierer, der eine gegebene Simulationsbibliothek verwendet, irrelevant, da die Prozessimplementation aus seiner Sicht bereits fertig im Simulationskern der Bibliothek vorliegt.

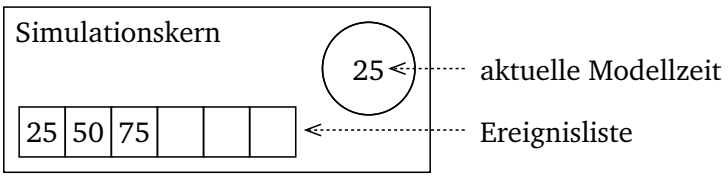

Abbildung 2.9: Aufbau eines sequentiellen Simulationskerns

Implementation des Simulationskerns

Die Implementation eines Simulationskerns ist vergleichsweise einfach (siehe Abbildung 2.9): Als statische Komponenten besitzt ein Simulationskern im Prinzip nur eine Variable, die während einer Simulation stets die aktuelle Modellzeit enthält, und einen sogenannten *Ereigniskalender* (*schedule* oder auch *event list*). In letzteren werden während eines Simulationslaufes alle bis zum jeweiligen Modellzeitpunkt bereits bekannten, zukünftigen Ereignisse sortiert nach ihrem Eintrittszeitpunkt eingetragen (in Abbildung 2.9 sind von den Ereignissen lediglich die Eintrittszeitpunkte dargestellt).

Während bei einem ereignisorientierten Simulationsmodell die identifizierten Ereignisse direkt im Ereigniskalender eingetragen werden können, werden bei prozessorientierten Modellen Prozessaktivierungen im Ereigniskalender notiert. Im Implementationskontext werden auch diese Prozessaktivierungen Ereignisse genannt. Wenn im Folgenden nicht anders vermerkt, kann mit *Ereignis* sowohl ein „echtes“ Ereignis im Sinne der ereignisorientierten Modellierung als auch eine Prozessaktivierung gemeint sein. Analog kann *Ausführung einer Ereignisroutine* im Rest dieser Arbeit auch für die Ausführung des Lebenszyklus' eines Prozesses bis zu dessen nächster Unterbrechung stehen.

Vor einem Simulationslauf wird ein Initialzustand des Simulationsmodells durch eine Initialbelegung der Zustandsvariablen hergestellt. Des Weiteren wird der Ereigniskalender mit initialen Ereignissen gefüllt. Der Simulationslauf selbst erfolgt dann über eine einfache Schleife, die in jedem Durchlauf:

1. das früheste Ereignis bzgl. Modellzeit aus dem Ereigniskalender holt,
2. die Modellzeit des Simulators auf den Eintrittszeitpunkt dieses Ereignisses vorstellt und
3. die Ereignisroutine des Ereignisses ausführt. Diese Ausführung beinhaltet:
   - die Änderung des Modellzustandes,
   - die Ausgabe/Speicherung von Informationen über das aktuelle Ereignis bzw. den aktuellen Zustand für eine spätere Auswertung sowie
   - die Ermittlung zukünftiger Ereignisse und deren Eintragung im Ereigniskalender.

Der Simulationslauf ist beendet, wenn vor der Bearbeitung eines Ereignisses mindestens eine der folgenden Bedingungen eintritt:

- der Ereigniskalender enthält keine Ereignisse mehr,
- ein vorher festgelegter Endzustand wird erreicht,
- eine vorher festgelegte Endzeit (bzgl. Modellzeit) wird erreicht und/oder
- eine vorher festgelegte Endzeit (bzgl. Ausführungszeit) wird erreicht.[5]

Sobald ein Simulationslauf beendet ist, dienen die während der Abarbeitung der Ereignisroutinen erfolgten Ausgaben bzw. angelegten Daten als Grundlage für die Rückinterpretation der Simulationsergebnisse in das Ausgangsszenario.

[5] Dies ist sinnvoll bei Simulationen, die potentiell endlos laufen würden, ohne dass eine der vorher genannten Bedingungen erreicht werden würde.

KAPITEL 3

GRUNDLAGEN PARALLELER SIMULATION

Die Hauptmotivation bei der Parallelisierung von diskreten ereignisgesteuerten Simulationen liegt in einer angestrebten schnelleren Durchführung von Simulationsläufen. Analog zur Parallelisierung von Algorithmen in anderen Anwendungsdomänen besteht dabei die Grundidee in der Identifikation und anschließenden parallelen Durchführung voneinander unabhängiger Berechnungen. Ebenso analog zu Parallelisierungsansätzen in anderen Anwendungsfeldern besteht jedoch auch hier prinzipiell die Gefahr, dass zur Laufzeit ein parallelisierungsbedingter Zusatzaufwand die erreichten Geschwindigkeitssteigerungen wieder zunichte macht.

Zwei weitere, nicht zu unterschätzende Aspekte bei der Parallelisierung von Simulationsalgorithmen sind der potentielle zusätzliche Aufwand bei der Erstellung des Simulationsprogrammes, insbesondere bei der Implementation des Simulationskerns, sowie eventuell mit der Parallelisierung verbundene Einschränkungen für den Simulationsmodellierer bei der Erstellung passender Simulationsmodelle.

Im Folgenden werden verschiedene Parallelisierungsansätze vorgestellt und die jeweils damit verbundenen Vor- und Nachteile kurz diskutiert. Anschließend wird der in dieser Arbeit verwendete Parallelisierungsansatz detaillierter beschrieben, bevor seine Umsetzung und die dabei auftretenden Probleme sowie zugehörige Lösungsansätze vorgestellt werden.

3.1 Parallele vs. verteilte Simulation

Es gibt in der Simulationsliteratur verschiedene Auffassungen, worin der Unterschied zwischen verteilter und paralleler Simulation besteht. In dieser Arbeit wird die Sichtweise Fujimotos [Fuj99] verwendet, der eine Grenze zwischen beiden Simulationsarten anhand der eingesetzten Hardware zieht.

Von *paralleler Simulation* spricht man dann, wenn ein Simulationslauf parallelisiert „innerhalb“ eines Rechners mit mehreren Recheneinheiten (Prozessoren bzw. Prozessorkernen) stattfindet. Dabei spielt die konkrete Architektur (*shared-memory*, *distributed-memory* oder *single instruction multiple data (SIMD)*) für die Einordnung keine Rolle. Eine *verteilte Simulation* setzt hingegen mehrere, durch ein Netzwerk miteinander verbundene Rechner voraus, die geographisch beliebig verteilt sein dürfen.

Für den Simulationskernentwickler ist diese Unterscheidung insofern wichtig, als dass im Falle einer verteilten Simulation zusätzlich netzwerkspezifische Effekte, z. B. die Verzögerung oder der Verlust von Nachrichten, zwischen den einzelnen Recheneinheiten berücksichtigt werden müssen. Da es sich dabei ausschließlich um zusätzliche Anforderungen gegenüber einer parallelen Simulation handelt, ergibt sich, dass jeder verteilte Simulationsalgorithmus prinzipiell auch als ein „lediglich“ paralleler benutzt werden kann. Dennoch bietet sich auf Einzelcomputern die Verwendung von nicht verteilten, parallelen Simulationsalgorithmen an, da diese durch den Wegfall des netzwerkspezifischen Zusatzaufwandes schneller arbeiten.

Zusätzlich kann ein rein paralleler Simulationsalgorithmus konkrete Architekturmerkmale des eingesetzten Rechners zur Simulationsbeschleunigung ausnutzen. Beispielsweise kann bei Rechnern mit geteiltem Speicher für alle Recheneinheiten (*shared memory*) dieser als schneller alternativer Kommunikationsweg zwischen selbigen benutzt werden.

In der vorliegenden Arbeit werden ausschließlich parallele Simulationsalgorithmen betrachtet. Auch bei den im Folgenden vorgestellten, klassischen verteilten Simulationsalgorithmen werden alle Spezialfälle ignoriert, die sich nur durch netzwerkspezifische Probleme ergeben können. Insbesondere wird in dieser Arbeit die Kommunikation zwischen verschiedenen Recheneinheiten als fehler- und verzögerungsfrei betrachtet.

3.2 Möglichkeiten der Parallelisierung von Simulationen

Parallele Ausführung von Simulationsläufen

Die Erkenntnis, dass bei einer Vielzahl von Simulationen nicht nur ein einzelner Simulationslauf, sondern eine ganze Serie von Simulationsläufen beschleunigt werden soll, führte zur wohl einfachsten Form der Parallelisierung: der parallelen Durchführung von Simulationsläufen sequentieller Simulationen ([BDO85], [Vak92], [GP01]).

Dieser Ansatz ist besonders dann sinnvoll, wenn eine große Anzahl voneinander unabhängiger Simulationsläufe durchgeführt werden soll. Dies kommt insbesondere bei Simulationen mit stochastischen Elementen vor, wo die Simulationsresultate erst ab einer gewissen Anzahl von Simulationsläufen eine notwendige statistische Signifikanz erhalten.

Ein weiterer Anwendungsfall ist die Suche nach optimalen Werten für Simulationsparameter durch ein iteratives Durchführen von Simulationsläufen, wobei jedes erhaltene Resultat zu einer Korrektur der Parameterbelegungen im folgenden Simulationslauf verwendet wird. Obwohl in diesem Fall einzelne Simulationsläufe direkt voneinander abhängig sind, lässt sich dennoch die insgesamt benötigte Zeit reduzieren, wenn in jeder Verfeinerungsstufe statt nur eines Simulationslaufes gleich mehrere mit unterschiedlichen, potentiell besseren Parameterbelegungen gestartet werden ([GP01]).

Der offensichtliche Vorteil der Parallelisierung von kompletten Simulationsläufen liegt darin, dass bei der Implementation kein Mehraufwand gegenüber der Umsetzung einer sequentiellen Simulation betrieben werden muss. Der ebenso offensichtliche Nachteil besteht jedoch in der nicht vorhandenen Beschleunigung eines einzelnen Simulationslaufes, so dass dieses Verfahren nutzlos ist, wenn wahlweise nur einzelne oder aber voneinander abhängige Simulationsläufe durchgeführt werden müssen.

Parallelisierung von simulationsfernen Berechnungen

Ein weiterer Parallelisierungsansatz ist die Parallelisierung simulationsferner Berechnungen (Zufallszahlengenerierung, Statistik, ...) innerhalb einzelner Ereignisroutinen ([Com84], [WSY83]). Wenn während jedes Simulationslaufes sichergestellt wird, dass jede zu einer Ereignisroutine gestartete, parallelisierte Berechnung spätestens mit der Beendigung der Ereignisroutine ebenfalls terminiert, so lässt sich dieser Ansatz mit einem unmodifizierten sequentiellen Simulationskern durchführen.

Trotz des geringen Implementationsaufwandes wird die Parallelisierung simulationsferner Berechnungen selten eingesetzt, da sich nur für sehr wenige Simulationsmodelle eine relevante Simulationslaufbeschleunigung ergibt [Meh94].

Parallelisierte Ausführung von Ereignisroutinen gleichzeitiger Ereignisse

Ein weiterer Ansatz zur Parallelisierung, der nur wenige Änderungen gegenüber einer sequentiellen Simulationsimplementation erfordert, ist die parallele Abarbeitung von Ereignisroutinen gleichzeitig auftretender Ereignisse ([Pre90], [Wil86]). Ein derartig parallelisierter Simulationskern basiert zwar weiterhin auf einer zentralen Schleife, die über einem Ereigniskalender iteriert. Allerdings werden in dieser Schleife in jedem Durchlauf nicht nur das früheste Ereignis, sondern mehrere bzw. im Idealfall alle Ereignisse mit dem gleichen Zeitstempel wie dem des frühesten Ereignisses parallel abgearbeitet. Anschließend muss sichergestellt werden, dass alle parallel gestarteten Ereignisroutinen einer Modellzeit beendet werden, bevor ein Ereignis einer späteren Modellzeit behandelt wird.

Auch dieses Verfahren hat sich nicht durchgesetzt, da in den meisten Modellen nur wenige Ereignisse mit identischen Zeitstempeln auftauchen und demzufolge nur

selten eine relevante Beschleunigung eintritt. Auch die in [Wil86] beschriebene Modifikation des Ansatzes durch die Erweiterung der Menge parallel auszuführender Ereignisroutinen um die Ereignisroutinen unmittelbar folgender Ereignisse hat an dieser Tatsache nur wenig geändert.

Parallelisierung durch Verteilung des Simulationsmodells

Der wirksamste und in der vorliegenden Arbeit ausschließlich betrachtete Parallelisierungsansatz besteht in der Aufteilung des Simulationsmodells in kooperierende Teilmodelle, die parallel durch separate Recheneinheiten ausgeführt werden ([Fuj99], [Meh94]). Dabei besteht der wesentliche Unterschied zu den bisher genannten Parallelisierungen darin, dass bei diesem Ansatz mitnichten nur gleichzeitig eintretende Ereignisse parallel bearbeitet werden. Stattdessen werden Ereignisse aus verschiedensten Modellzeiten parallel betrachtet, was einerseits einen hohen potentiellen Geschwindigkeitsgewinn ermöglicht, andererseits aber auch einen erhöhten Synchronisationsaufwand zur Sicherstellung eines korrekten Simulationsergebnisses erfordert.

3.3 Grundlagen der Simulation mit verteiltem Simulationsmodell

Nach [Meh94] müssen drei zentrale Probleme gelöst werden, wenn eine Simulation durch den Ansatz der Verteilung des Simulationsmodells parallelisiert werden soll: die Partitionierung des Simulationsmodells, die Zuordnung der Teilmodelle auf Recheneinheiten und die Sicherung der Kausalität durch Synchronisation. Im Folgenden werden diese drei Probleme einzeln diskutiert.

Partitionierung des Simulationsmodells

Bei der Lösung des ersten Problems, der Aufteilung des Simulationsmodells, hat sich eine Modellierungssicht als zweckmäßig erwiesen, die auf die Arbeiten von Chandy und Misra [CM81, Mis86] zurückgeht. Die Grundlage dieser Modellierungssicht ist die Zerlegung eines Systems in Teilsysteme, die *physische Prozesse (PP)* genannt werden. Bei dieser Zerlegung müssen die folgenden Bedingungen erfüllt sein:

- jedes in einem physischen Prozess auftretende Ereignis darf nur Zustandsänderungen im selben physischen Prozess bewirken,
- die durch ein Ereignis ausgelösten Folgeereignisse dürfen sowohl im eigenen (interne Ereignisse) als auch in fremden (externe Ereignisse) physischen Prozessen stattfinden,
- zur Benachrichtigung über externe Ereignisse besitzen die physischen Prozesse die Möglichkeit, sich gegenseitig *Nachrichten* (*messages*) zu schicken und

- es gibt keinen Systemzustand außerhalb der physischen Prozesse, also insbesondere keine Zustandsvariablen auf globalem Niveau.

Hat man ein derartiges, ausschließlich aus kommunizierenden, physischen Prozessen zusammengesetztes Simulationsmodell vorliegen, so lässt sich eine Simulation desselbigen folgendermaßen umsetzen: Jeder physische Prozess wird in der Implementation in einer separaten Programmkomponente, einem sogenannten *logischen Prozess (LP)* umgesetzt. Die Implementation eines solchen LPs ist bis auf eine Ausnahme mit der in Abschnitt 2.8 beschriebenen Implementation eines ereignisgesteuerten Simulationskerns identisch. Insbesondere besitzt jeder LP seinen eigenen Ereigniskalender und seine eigene Uhr bzgl. Modellzeit.

Die Ausnahme betrifft die Erzeugung von Ereignissen, die sich im Teilmodell eines anderen LPs befinden. In diesem Fall schickt der erste LP dem zweiten eine Nachricht, die das von letzterem einzuplanende Ereignis enthält, und bildet dadurch genau die Nachricht nach, die im Originalsystem der erste PP dem zweiten geschickt hätte.

An dieser Stelle soll kurz auf die Bezeichnungen *physischer Prozess* und *logischer Prozess* eingegangen werden. Trotz der Namensähnlichkeit haben diese prinzipiell nichts mit den in Abschnitt 2.8 vorgestellten Prozessen der prozessorientierten Modellierungssicht gemein. Insbesondere der Begriff *logischer Prozess* ist problematisch, da er keinen Prozess als Modellbestandteil, sondern einen Simulationskern bezeichnet. Zur Vermeidung von Verwechslungen werden in dieser Arbeit bevorzugt die Abkürzungen *PP* (Plural: *PPs*) für einen physischen und *LP* (Plural: *LPs*) für einen logischen Prozess benutzt.

Zuordnung von LPs auf Recheneinheiten

Das zweite zu lösende Problem ist eine geeignete Zuordnung von LPs auf vorhandene Recheneinheiten. Im Idealfall kann jedem LP eine eigene Recheneinheit zugewiesen werden, so dass alle LPs vollkommen parallel arbeiten können. Stehen hingegen nicht ausreichend viele Recheneinheiten zur Verfügung, müssen einzelne Recheneinheiten mehrere LPs abwechselnd ausführen. Dabei besteht die Wahl zwischen einer statischen Verteilung der LPs auf die Recheneinheiten zum Simulationsbeginn oder aber einer dynamischen Verteilung während der Simulationslaufzeit.

Je nach verwendeter Hardware, konkretem Simulationsmodell und gewünschtem Optimierungsgrad kann dieses zweite Problem sehr kompliziert werden, z. B. bei der Betrachtung unterschiedlich leistungsfähiger Recheneinheiten oder der Berücksichtigung eines unterschiedlichen Kooperationsaufwandes zwischen einzelnen Recheneinheiten.

In dieser Arbeit wird das Problem der Zuordnung von LPs auf Recheneinheiten weitgehend ausgeblendet. Eine kurze Betrachtung der gewählten, vergleichsweise pragmatischen Lösung befindet sich in der Implementationsbeschreibung in Abschnitt 5.3.

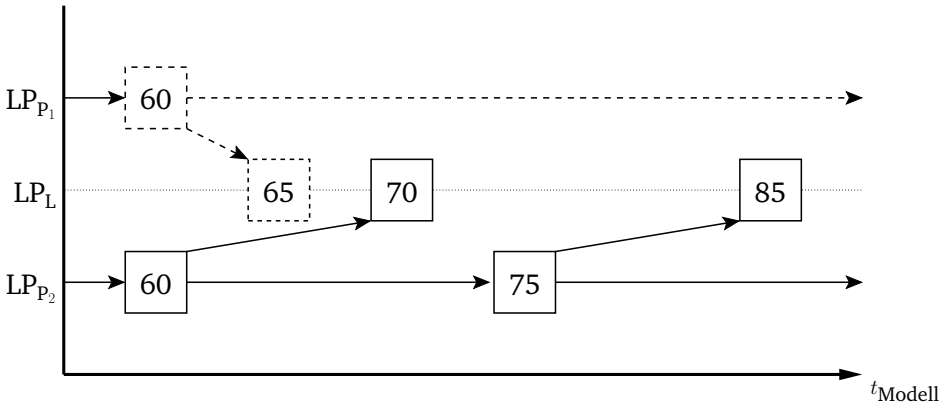

Abbildung 3.1: Beispiel eines Kausalitätsfehlers

Kausalität und Kausalitätsfehler

Das dritte Problem ist die Sicherstellung, dass trotz der gleichzeitigen (bzgl. Ausführungszeit) Ausführung nichtgleichzeitiger (bzgl. Modellzeit) Ereignisse korrekte Simulationsergebnisse erzeugt werden.

Wie beschrieben, arbeitet während eines Simulationslaufes jeder LP die ihm bekannten Ereignisse entsprechend der Zeitstempel nacheinander ab und schreitet dabei in der Modellzeit voran. Als Seiteneffekt der Ereignisbearbeitung erzeugt ein LP zukünftige Ereignisse sowohl bei sich selbst (interne Ereignisse) als auch bei fremden LPs (externe Ereignisse).

Von einem *Kausalitätsfehler* spricht man, wenn ein LP eine Nachricht über ein Ereignis erhält, dessen Zeitstempel bezüglich der aktuellen Modellzeit des LPs in der Vergangenheit liegt. Der von diesem LP bereits errechnete Simulationsmodellzustand wird in diesem Moment ungültig, da bei dessen Berechnung das nachträglich bekannt gewordene Ereignis unberücksichtigt blieb. Ein derartiges „verspätetes“ Ereignis nennt man *Nachzüglerereignis* (*straggler event*); die Nachricht, die das Nachzüglerereignis referenziert, bezeichnet man als *Nachzüglernachricht* (*straggler message*).

Angenommen, das Beispielszenario der Spielzeugfabrik aus Abschnitt 2.5 wäre so modelliert und partitioniert, dass jede Station durch einen eigenen LP simuliert wird. Da in diesem Fall die beiden LPs der Produktionsstationen LP_{P_1} und LP_{P_2} unabhängig in der Modellzeit voranschreiten können, ist es möglich, dass der LP der Lackierstation LP_L bereits Ankunftsereignisse der „schnelleren“ (bzgl. Ausführungszeit) Produktionsstation abgearbeitet hat, wenn ihr ein eigentlich früheres Ankunftsereignis der langsameren Produktionsstation bekannt wird.

Abbildung 3.1 soll dies verdeutlichen, indem es einen Ausschnitt aus einem möglichen Simulationslauf zeigt. Dargestellt sind die LPs der drei Stationen mit zugehörigen Ereignissen, wobei die Pfeile zwischen den Ereignissen anzeigen, welches Ereignis von welchem erzeugt wurde. Angenommen, zu einem bestimmten Zeitpunkt bzgl. der Ausführungszeit sind die einzelnen LPs derart in der Modellzeit vor-

angeschritten, dass die mit durchgezogenen Linien dargestellten Ereignisse bereits abgearbeitet wurden, während die gestrichelten Ereignisse den LPs noch unbekannt sind. Wenn nun LP_{P_1} das Ereignis zur Zeit 60 erhält, bearbeitet und im Zuge dessen LP_L über das Ereignis zum Zeitpunkt 65 informiert, entsteht ein Kausalitätsfehler, da LP_L bereits die späteren Ereignisse zu den Zeitpunkten 70 und 85 behandelt hat. Das Ereignis zur Zeit 65 ist in diesem Fall das Nachzüglerereignis.

Ist eine parallele Simulation frei von Kausalitätsfehlern, so spricht man davon, dass die *Kausalität* gewahrt wurde; das zentrale Problem der Verhinderung von Kausalitätsfehlern wird in der Simulationsliteratur *Synchronisationsproblem* genannt. Dabei gibt es zwei prinzipielle Möglichkeiten, mit dem Synchronisationsproblem umzugehen. Entweder man verhindert Kausalitätsfehler von vornherein durch Synchronisation der einzelnen LPs oder aber man toleriert diese Fehler zunächst und korrigiert nachträglich die durch sie verursachten Inkonsistenzen. Der erste Ansatz führte zur Entwicklung der in Abschnitt 3.4 beschriebenen konservativ-parallelen Simulation, während der zweite die Grundlage der in Abschnitt 3.5 diskutierten optimistisch-parallelen Simulation darstellt.

3.4 Konservativ-parallele Simulation

In der sogenannten *konservativ-parallelen Simulation* werden Kausalitätsfehler von vornherein durch Synchronisation ausgeschlossen. Dabei besteht die Grundidee darin, dass während eines Simulationslaufes jeder LP nur dann in der Modellzeit voranschreiten darf, wenn er dadurch nicht Gefahr läuft, Nachzüglerereignisse zu erhalten.

Es gibt verschiedene Simulationsalgorithmen, die der konservativ-parallelen Simulation zuzuordnen sind. Der wohl bekannteste ist der *Chandy-Misra-Bryant-Algorithmus* (auch als *Null-Message-Algorithmus* bekannt) [CM81, Mis86], der im Folgenden exemplarisch vorgestellt wird, um anschließend mit seiner Hilfe weitere Grundbegriffe der parallelen Simulation einzuführen.

Der Chandy-Misra-Bryant-Algorithmus

Die Grundlage für den Algorithmus bildet eine Menge von LPs, die sich durch Nachrichtenaustausch gegenseitig Ereignisse einplanen können. Dabei gelten zusätzlich die folgenden zwei Einschränkungen:

- die Zeitstempel der von einem LP generierten Ereignisse dürfen in ihrem Wert nie abnehmen und
- bereits zu Simulationsbeginn kennt jeder LP alle LPs, von denen er auch nur potentiell Nachrichten empfangen wird.

Die erste Einschränkung ermöglicht einem LP die Entscheidung, wie weit er in der Modellzeit gegenüber einem konkreten anderen LP fortschreiten darf: Sobald er von einem LP eine Nachricht über ein Ereignis erhält, kann er gefahrlos in der

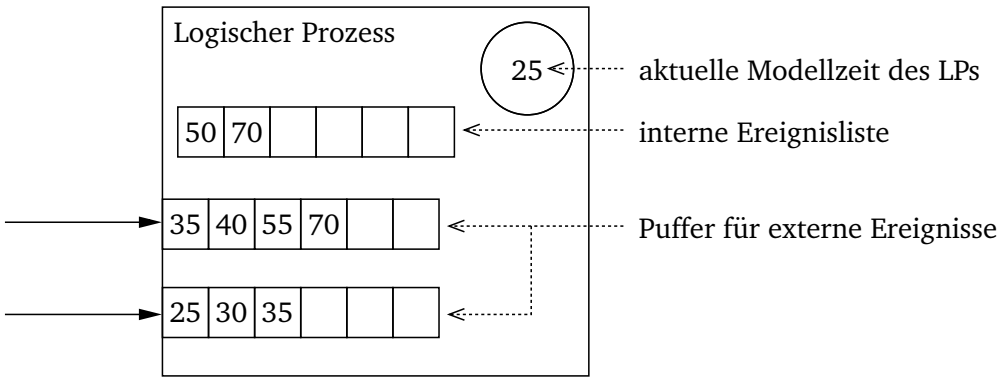

Abbildung 3.2: Aufbau eines LPs beim Chandy-Misra-Bryant-Algorithmus

Modellzeit bis zu dessen Eintrittszeitpunkt voranschreiten. Die zweite Einschränkung ermöglicht die Erweiterung dieser Erkenntnis auf die Menge aller potentiell (Nachzügler-)Nachrichten sendenden LPs: Sobald ein LP von jedem ihm bekannten LP mindestens eine Nachricht empfangen hat, ist es sicher, bis zur Modellzeit des Ereignisses mit dem frühesten Eintrittszeitpunkt voranzuschreiten.

In der Implementation ist ein LP beim Chandy-Misra-Bryant-Algorithmus zunächst wie ein sequentieller Simulationskern aufgebaut (siehe Abbildung 3.2): Er besitzt eine eigene Uhr bzgl. Modellzeit und einen Puffer für Ereignisse, die er selbst generiert hat. Zusätzlich stellt er für jeden ihm bekannten LP, also jeden LP, von dem er potentiell Nachrichten empfangen wird, einen separaten Ereignispuffer zur Verfügung. In diesem Ereignispuffer werden die in den empfangenden Nachrichten enthaltenen Ereignisse gespeichert. Sowohl die Ereignisse in der internen Ereignisliste als auch die in den Ereignispuffern sind nach ihren Eintrittszeitpunkten geordnet.

Die Arbeitsweise eines derartig aufgebauten LPs weicht kaum von der eines sequentiellen Simulationskerns ab (vgl. Abschnitt 2.8). Der erste wesentliche Unterschied besteht darin, dass in jeder Iteration der zentralen Schleife zuerst unter allen in den Puffern gespeicherten Ereignissen das Ereignis mit dem frühesten Eintrittszeitpunkt ermittelt werden muss, bevor dieses dann zum Fortschreiten der Simulation ausgewertet wird.

In Abbildung 3.3 ist ein Ausschnitt einer möglichen Modellierung des Beispielszenarios aus Abschnitt 2.5 für den Chandy-Misra-Bryant-Algorithmus dargestellt. Man kann gut erkennen, dass die LPs der Produktionsstationen gar keine Puffer für externe Nachrichten besitzen, da sie von keinen anderen LPs Nachrichten zu erwarten haben. Der LP der Lackierstation hingegen besitzt für die eintreffenden Nachrichten der LPs der beiden Produktionsstationen je einen Puffer, während der interne Puffer aufgrund des Verhaltens der Lackierstation nie gefüllt werden wird.

Problematisch wird der Chandy-Misra-Bryant-Algorithmus, wenn einer der externen Puffer eines LPs „leerläuft“, da in diesem Fall der entsprechende LP anhalten muss, bis wieder eine Nachricht des entsprechenden Senders eintrifft. Da dies unter Umständen nie passiert, z. B. wenn zwei LPs gegenseitig aufeinander warten,

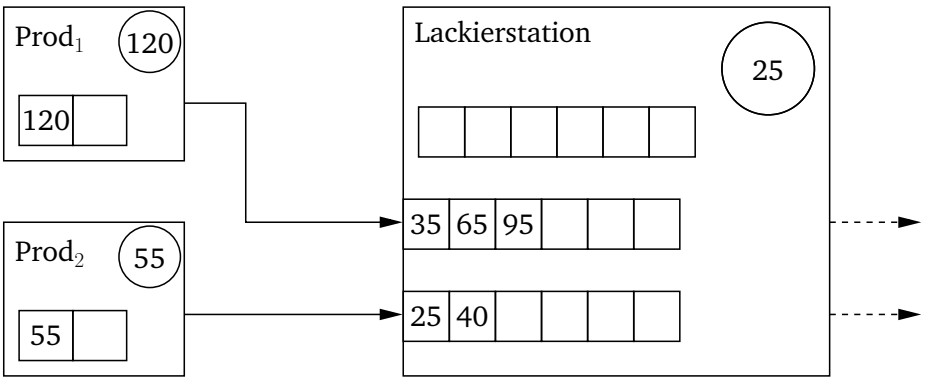

Abbildung 3.3: Mögliche Verteilung des Simulationsmodells des Beispielszenarios im Chandy-Misra-Bryant-Algorithmus

wurde im Chandy-Misra-Bryant-Algorithmus das Konzept der *Leernachrichten* (*null messages*) eingeführt.

Eine Leernachricht ist eine Nachricht, die nicht zur Information über ein Ereignis verwendet wird, sondern lediglich der Übermittlung der aktuellen Modellzeit des sendenden LPs dient. Dazu enthält die Nachricht ein „leeres“ Ereignis, dessen Eintrittszeitpunkt mit der aktuellen Modellzeit des sendenden LPs übereinstimmt.

Ändert man den bisher beschriebenen Algorithmus so ab, dass jeder eine Nachricht verschickende LP gleichzeitig an alle weiteren, ihn kennenden LPs je eine Leernachricht sendet, so wird der Fall eines leerlaufenden Eingangspuffers nie auftreten. Stattdessen werden sich im Falle fehlender „echter“ Nachrichten genügend Leernachrichten in den externen Puffern der LPs befinden, so dass diese fortschreiten können.

Als Alternative zu dieser, auch als *Push-Verfahren* bezeichneten Herangehensweise existiert das sogenannte *Pull-Verfahren*. Bei diesem werden Leernachrichten nur auf explizite Anforderung eines LPs mit „leergelaufenem“ Eingangspuffer verschickt. Beide Ansätze haben ihre Vor- und Nachteile: Während beim Push-Verfahren eine große Menge unnötiger Nachrichten verschickt wird, erfordert das Pull-Verfahren wahlweise einen weiteren Nachrichtenkanal oder aber einen zusätzlichen Nachrichtentyp zur Anforderung von Leernachrichten. Weitere Informationen zu beiden Ansätzen inklusive einer detaillierten Abhandlung der Vor- und Nachteile befinden sich in [Fuj99].

Lookahead

Eine Verbesserung des Chandy-Misra-Bryant-Algorithmus, die auch bei anderen konservativ-parallelen Simulationsalgorithmen funktioniert, ist die Berücksichtigung von sogenannten *Lookaheads*. Unter einem Lookahead zwischen zwei LPs versteht man einen modellinhärenten, minimalen Abstand zwischen der aktuellen Modell-

zeit eines sendenden LPs und dem Zeitstempel des in der Nachricht referenzierten Ereignisses.

Im Beispielszenario aus Abschnitt 2.5 benötigt ein in der Produktionsstation P_1 hergestellter Würfel stets fünf Sekunden, bis er an der Lackierstation angekommen ist. In eine parallele Simulation übertragen bedeutet dies, dass alle Nachrichten, die zum Zeitpunkt t_0 vom LP_{P_1} an LP_L geschickt werden, nur Ereignisse enthalten können, deren Eintrittszeitpunkt $t_0 + 5$ beträgt. Demzufolge besitzt der Kommunikationskanal zwischen den beiden LPs einen Lookahead von fünf Sekunden.

Ist ein derartiger Lookahead bekannt, kann er im Chandy-Misra-Bryant-Algorithmus leicht zur Optimierung benutzt werden, indem die „leeren Ereignisse“ in Leernachrichten nicht mehr die aktuelle Modellzeit des sendenden LPs, sondern stattdessen die aktuelle Modellzeit plus dem Lookahead als Eintrittszeitpunkt erhalten. Damit kann der die Leernachricht erhaltende LP sofort weiter in der Modellzeit voranschreiten und dabei eventuell unnötige Blockaden auslassen, die im Pull-Verfahren mit zusätzlichen Leernachrichtanforderungen verbunden wären.

Aufgrund ihrer Modellabhängigkeit sind Lookaheads stark von Modelländerungen betroffen. Insbesondere besteht stets die Gefahr, dass ein bereits ermittelter Lookahead durch eine solche Änderung bis auf den Wert null reduziert wird. Daher sollte vor jeder aufwendigeren Ermittlung eines Lookaheads der voraussichtlich damit verbundene Aufwand in Hinblick auf zu erwartende Modelländerungen überprüft werden.

Weitere konservativ-parallele Verfahren

Neben dem Chandy-Misra-Bryant-Algorithmus existieren weitere Ansätze zur konservativ-parallelen Simulation, die auch kombiniert auftreten können [Meh94]. Allen gemein ist die Grundidee, die einzelnen LPs während eines Simulationslaufes gezielt so „abzubremsen“, dass ein LP niemals eine Nachzüglernachricht erhalten kann. Damit verbunden ist die Fragestellung, wie die dabei benötigten „sicheren“ Modellzeiten berechnet werden können.

Während der Chandy-Misra-Bryant-Algorithmus derartige Modellzeiten implizit während der Arbeit seiner LPs berechnet, wird in alternativen Ansätzen diese Berechnung explizit durchgeführt. Infolgedessen kann ein LP in diesen Verfahren wieder wie ein sequentieller Simulationskern, also auf der Grundlage nur eines Puffers für alle Nachrichten/Ereignisse, implementiert werden. Dafür enthalten derartige LPs eine modifizierte zentrale Schleife, die vor jeder Bearbeitung eines Ereignisses prüft, ob der LP durch dessen Bearbeitung eine unsichere Modellzeit erreicht. Verschiedene Varianten der Berechnung sicherer Modellzeiten und deren Verteilung an die einzelnen LPs befinden sich in [Fuj99].

Analyse und Bewertung konservativ-paralleler Verfahren

In Abbildung 3.4 sind Laufzeitdiagramme einer sequentiellen und einer konservativ-parallelen Simulation mit drei LPs zu sehen. Das Laufzeitdiagramm der parallelen

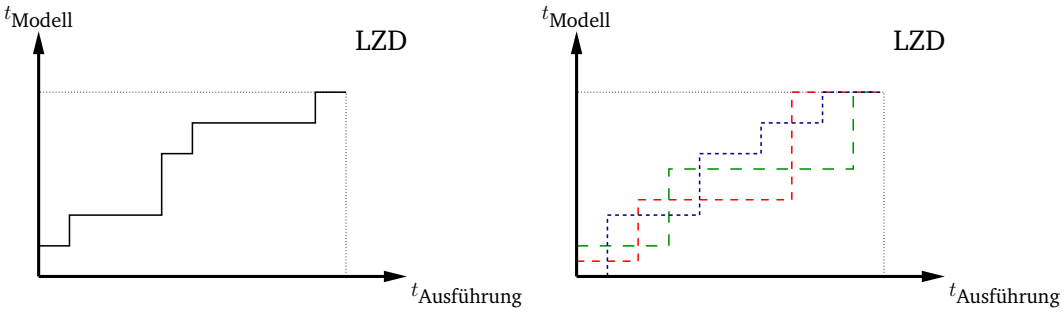

Abbildung 3.4: Laufzeitdiagramme eines sequentielles und einer konservativ-parallelen Simulationslaufes

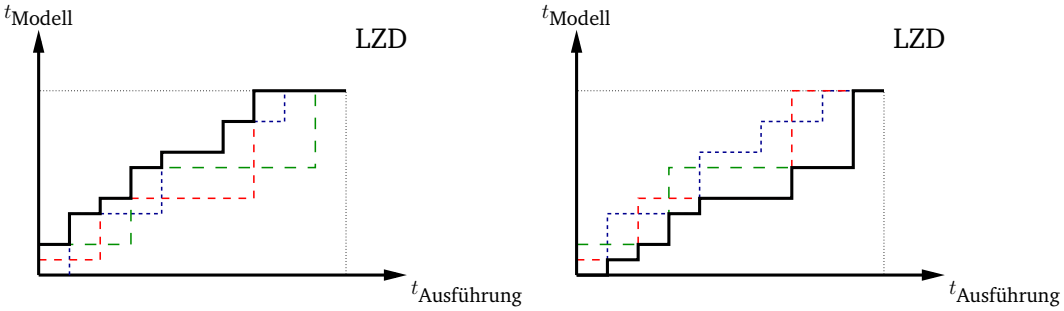

Abbildung 3.5: Laufzeitdiagramme eines konservativ-parallelen Simulationslaufes mit Hervorhebung der minimalen und maximalen Modellzeiten

Simulation enthält mehrere Kurven, da jeder LP einen separaten sequentiellen Simulationskern darstellt. Eine Simulation ist in diesem Fall erst dann beendet, wenn der letzte LP die finale Modellzeit erreicht hat. Eine Gemeinsamkeit des sequentiellen Simulationskerns und der LPs besteht darin, dass deren Modellzeit ausschließlich zunehmen oder stagnieren, aber nie rückläufig sein kann.

Um während eines Simulationslaufes Auskunft über den Fortschritt einer konservativ-parallelen Simulation zu erhalten, bieten sich zwei Maße an: das Minimum und das Maximum der Modellzeiten aller LPs. In Abbildung 3.5 ist zweimal das Laufzeitdiagramm der konservativ-parallelen Simulation aus Abbildung 3.4 zu sehen, wobei jeweils ein zusätzlicher Graph zu jedem Modellzeitpunkt das Maximum bzw. Minimum der Modellzeiten aller LPs verdeutlicht. Dabei ist die minimale Modellzeit besonders interessant, da diese zu jedem Zeitpunkt (bezüglich Ausführungszeit) eine sichere Modellzeit darstellt, d. h., der LP mit der minimalen Modellzeit kann nie eine Nachzüglernachricht erhalten. Diese Tatsache ist letztlich durch das Kausalitätsprinzip begründet, demzufolge jeder LP prinzipiell nur Ereignisse in der Zukunft bezüglich seiner aktuellen Modellzeit auslösen kann. Daher kann nie ein LP, dessen Modellzeit gegenüber einem anderen LP in der Zukunft liegt, letzterem eine Nachzüglernachricht schicken. Infolgedessen kann der LP mit der geringsten Modellzeit das aktuelle Ereignis stets gefahrlos abarbeiten.

Wie detailliert in [Fuj99] und in [Meh94] beschrieben, kann eine konservativ-parallele Simulation nur selten das parallele Potential eines Simulationsmodells ausnutzen. Der primäre Grund dafür liegt darin, dass eine konservativ-parallele Simulation immer bereits dann eine parallele Abarbeitung verhindert, wenn ein Kausalitätsfehler lediglich auftreten *könnte*.

Das größte Optimierungspotential konservativ-paralleler Simulationen liegt in der Ermittlung möglichst hoher Lookaheads. Da diese jedoch modellabhängig und im Allgemeinen nicht automatisch ermittelbar sind, ist es der Simulationsmodellierer, dem diese Aufgabe obliegt. Zusätzlich besteht insbesondere bei der Weiterentwicklung von Modellen das Problem, dass schon kleine Modelländerungen manuell ermittelte Lookaheads invalidieren können.

Ein Vorteil der konservativ-parallelen Simulation, insbesondere im Vergleich zur im Folgenden beschriebenen optimistisch-parallelen Simulation, besteht dafür in der guten Nachvollziehbarkeit der parallelen Simulationsläufe und der dadurch deutlich erleichterten Entwicklung von Simulationsmodellen.

3.5 Optimistisch-parallele Simulation

Die Grundidee der *optimistisch-parallelen* Simulationsalgorithmen liegt, im Gegensatz zu den konservativ-parallelen Verfahren, in der bewussten Tolerierung von Kausalitätsfehlern und deren späterer Korrektur. Damit wird der größte Kritikpunkt an den konservativ-parallelen Algorithmen, eine vorhandene Parallelität des Simulationsmodells nicht oder nicht genügend auszunutzen, von vornherein eliminiert: Eine optimistisch-parallele Simulation läuft immer parallel; unter Umständen selbst dann, wenn das Simulationsmodell eine solche Parallelität eigentlich ausschließt. Der Preis für diese stets garantierte Parallelität besteht in einem deutlich erhöhten Aufwand, der zur Korrektur ungültiger Simulationsmodellzustände betrieben werden muss.

Das wohl bekannteste und wichtigste optimistisch-parallele Simulationsverfahren ist der *Time-Warp-Algorithmus*. Er wurde erstmals von Jefferson beschrieben [JS82, Jef85] und im *Time-Warp operating system* implementiert [JBW+87]. Im Folgenden werden die Arbeitsweise und die dafür benötigten Komponenten beschrieben.

Aufbau und Arbeitsweise eines Time-Warp-LPs

Wie bei den konservativ-parallelen Simulationsalgorithmen hat sich auch bei den optimistisch-parallelen Verfahren die Modellierungssicht von Chandy und Misra (ein Simulationsmodell besteht aus kommunizierenden LPs – vgl. Abschnitt 3.3) durchgesetzt. Dabei ist ein LP im Time-Warp-Verfahren (in der Simulationsliteratur mitunter auch als *time warp logical process (TWLP)* bezeichnet) zunächst ähnlich einem sequentiellen Simulationskern aufgebaut (siehe Abbildung 3.6): Er besitzt eine Zeitvariable und einen Nachrichtenpuffer, wobei letztgenannter der Ereignisliste sequentieller Simulationskerne bzw. konservativ-paralleler LPs entspricht. Die Nach-

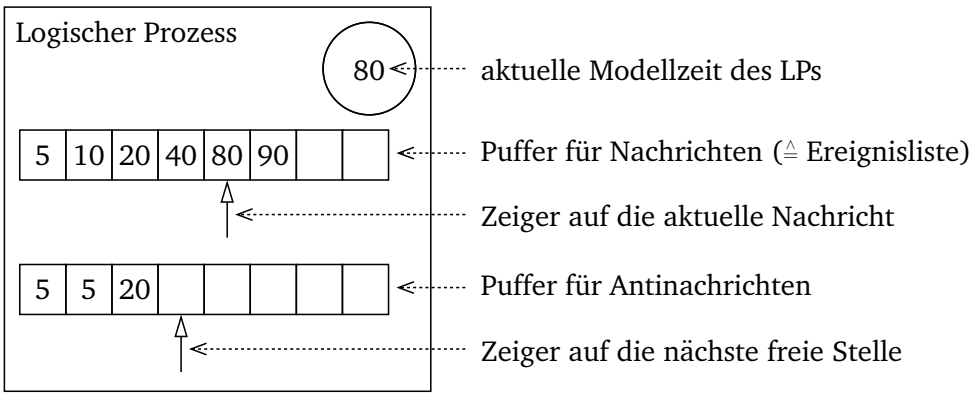

Abbildung 3.6: Aufbau eines LPs beim Time-Warp-Algorithmus

richten im Nachrichtenpuffer sind nach den Eintrittszeitpunkten der in den Nachrichten enthaltenen Ereignisse geordnet.

Damit die von einem LP für sich selbst erzeugten Ereignisse nicht aufwendig separat behandelt werden müssen, werden auch diese prinzipiell als Nachrichten verpackt und vom LP an sich selbst verschickt. Der Nachrichtenpuffer eines TWLPs speichert also sowohl interne als auch externe Ereignisse. Dabei enthält jede Nachricht im Time-Warp-Verfahren mindestens folgende Informationen:

- eine eindeutige Identifikationsnummer,
- eine Referenz auf den sendenden LP,
- eine Referenz auf den empfangenden LP,
- die aktuelle Modellzeit des sendenden LPs zum Sendezeitpunkt,
- den Eintrittszeitpunkt des Ereignisses und
- eine Beschreibung des Ereignisses bzw. eine Referenz auf ein Ereignisobjekt.

Im Gegensatz zu sequentiellen Simulationskernen und konservativ-parallelen LPs werden beim Time-Warp-Algorithmus die von einem LP abgearbeiteten Nachrichten nicht gelöscht, sondern verbleiben im Nachrichtenpuffer. Da sich demzufolge die als nächstes zu behandelnde Nachricht nicht mehr als erste Nachricht im Nachrichtenpuffer ermitteln lässt, besitzt jeder LP im Time-Warp-Verfahren einen Zeiger, der auf die jeweils aktuelle Nachricht im Nachrichtenpuffer zeigt. Während einer Simulation wird dieser Zeiger nach jeder Abarbeitung einer Nachricht um ein Feld des Nachrichtenpuffers weiter bewegt. Neben dem Nachrichtenpuffer für abzuarbeitende, also Ereignisse enthaltende Nachrichten besitzt jeder LP einen zweiten Nachrichtenpuffer für sogenannte Anti-Nachrichten. Dieser wird im folgenden Abschnitt erläutert.

Die grundsätzliche Arbeitsweise eines LPs ist analog zu der eines sequentiellen Simulationskerns: Entsprechend der Eintrittszeitpunkte der Ereignisse arbeitet er nacheinander die Nachrichten aus seinem Nachrichtenpuffer ab. Zusätzlich können sich LPs gegenseitig Nachrichten in ihren Nachrichtenpuffern eintragen.

Solange diese neu erhaltenen Nachrichten Ereignisse betreffen, deren Eintrittszeitpunkte bezüglich der aktuellen Modellzeit des betroffenen LPs in der Zukunft liegen, ergeben sich keine Probleme: Die erhaltenen Nachrichten werden einfach abgearbeitet, sobald der Zeiger des Nachrichtenpuffers bis zu ihnen vorgedrungen ist. Kritisch ist hingegen der Fall, wenn ein LP eine Nachricht über ein Ereignis erhält, dessen Eintrittszeitpunkt sich aus Sicht des LPs in der Vergangenheit befindet. Wie in Abschnitt 3.3 beschrieben, handelt es sich hierbei um einen klassischen Kausalitätsfehler, ausgelöst durch eine Nachzüglernachricht.

Der Time-Warp

Im Time-Warp-Algorithmus wird zur Korrektur eines Kausalitätsfehlers ein sogenannter *Time-Warp*[1] durchgeführt. Mit diesem aus dem Science-Fiction-Genre stammenden Begriff wird hier ein Zurückgehen in der Modellzeit eines LPs bezeichnet und zwar bis zum letzten Zeitpunkt, an dem der von diesem LP bearbeitete Simulationsmodellteil gültig war. Diesen Zeitpunkt in der Modellzeit bezeichnet man als *Rücksprungzeit*.

Damit ein Time-Warp modellweit den gewünschten Effekt der Kausalitätsfehlerbereinigung erzeugt, müssen drei Bedingungen erfüllt sein:

- die bereits vom LP bearbeiteten Ereignisse, deren Eintrittszeitpunkte hinter der Rücksprungzeit liegen, sich also nach dem Time-Warp wieder in der Zukunft des LPs befinden, müssen erneut abgearbeitet werden,
- die erfolgten ungültigen Zustandsänderungen im Simulationsmodell müssen rückgängig gemacht werden, d. h., der Modellzustand zur Rücksprungzeit muss wiederhergestellt werden und
- alle seit der Rücksprungzeit vom rücksetzenden LP an andere LPs versandten Nachrichten müssen revidiert werden.

Dank des beschriebenen Aufbaus eines LPs ist der erste Punkt vergleichsweise einfach durchzuführen (siehe Abbildung 3.7): Es muss lediglich nach dem Einfügen der Nachzüglernachricht in den Nachrichtenpuffer der Zeiger des Nachrichtenpuffers auf eben diese zurückgesetzt werden, wobei die Uhr des LPs automatisch entsprechend angepasst wird. Dadurch werden alle bereits abgearbeiteten Nachrichten, die bezüglich der Rücksprungzeit in der Zukunft liegen, erneut bearbeitet, da sie jetzt auch wieder in der Zukunft bezüglich der Modellzeit des LPs liegen.

Der zweite Punkt, die Zurücksetzung des vom LP bearbeiteten Simulationsmodellteils, kann auf zwei verschiedene Arten gelöst werden. Der kompliziertere Ansatz besteht darin, sorgfältig jede Änderung des Simulationsmodellteils zu protokollieren und im Falle eines Time-Warps diese Änderungen in umgekehrt chronologischer Reihenfolge zurückzunehmen, bis die Rücksprungzeit erreicht ist. Der einfachere und in dieser Arbeit verfolgte Ansatz besteht darin, bereits während der regulären

[1] Wörtlich übersetzt: *Zeitschleife*. Gemeint ist jedoch (üblicherweise auch in der englischen Sprache) die sich aus der Existenz einer Zeitschleife ergebende Möglichkeit eines *Zeitsprunges*.

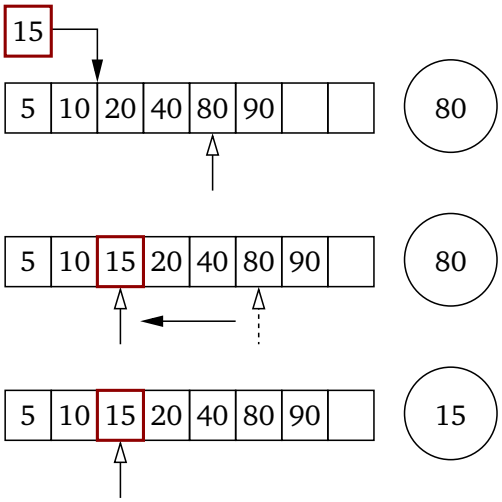

Abbildung 3.7: Ablauf eines Time-Warps im Nachrichtenpuffer eines LPs

Arbeit eines LPs nach jedem abgearbeiteten Ereignis eine Kopie des Simulationsmodellteils anzulegen. Wenn nun ein Time-Warp durchgeführt werden soll, muss lediglich die zur Rücksprungzeit gültige Kopie gesucht und im Folgenden als aktueller Simulationsmodellteil verwendet werden, während alle später bzgl. Modellzeit angelegten Modellteilkopien gelöscht werden können. Allerdings ist dieser Ansatz sehr speicherintensiv, was insbesondere bei großen Simulationsmodellen und/oder Simulationsläufen mit zahlreichen Ereignissen zu Problemen führen kann. Derartige Probleme werden inklusive möglicher Lösungsansätze in Abschnitt 3.5 diskutiert.

Um den dritten Punkt, das Zurückziehen inzwischen ungültig gewordener Nachrichten, umzusetzen, beinhaltet der Time-Warp-Algorithmus das Konzept der sogenannten *Anti-Nachrichten* (*anti-messages*). Dabei besteht die Grundidee darin, dass zu jeder Nachricht eine Anti-Nachricht existiert, die erstere aufheben kann. Umgesetzt wird diese Idee in der Implementation des Verhaltens der LPs bei der Ankunft von Anti-Nachrichten: Jeder LP, der eine Anti-Nachricht erhält, sucht umgehend in seinem Nachrichtenpuffer nach der zugehörigen originalen Nachricht und löscht anschließend sowohl die Nachricht als auch die Anti-Nachricht. Durch dieses Verhalten kann jeder LP eine bereits verschickte Nachricht zurückziehen, indem er demselben Empfänger eine entsprechende Anti-Nachricht hinterherschickt.

Im Idealfall trifft die Anti-Nachricht ein, bevor der empfangende LP die zugehörige Nachricht bearbeitet hat, so dass letztere problemlos aus dem Nachrichtenpuffer entfernt werden kann. Sollte hingegen die zurückzuziehende Nachricht bereits bearbeitet worden sein, so führt der betroffene LP ebenfalls einen Time-Warp aus. Dabei verwendet er den Eintrittszeitpunkt des zurückgezogenen Ereignisses als Rücksprungzeit. Es ist durchaus möglich, dass auch ein derartiger, durch einen Time-Warp ausgelöster, sekundärer Time-Warp weitere Time-Warps in weiteren LPs auslöst. Allerdings kann es nie zu endlosen Time-Warp-Kaskaden kommen, wie in

Abschnitt 3.5 noch gezeigt wird.

Damit der Umgang mit Anti-Nachrichten wie beschrieben funktioniert, sind gegenüber dem bereits erläuterten Aufbau noch einige Modifikationen nötig. Wie bereits in Abbildung 3.6 zu sehen war, besitzt ein LP neben dem Nachrichtenpuffer einen Puffer für Anti-Nachrichten. Wann immer ein LP eine Nachricht verschickt, generiert er gleichzeitig eine Anti-Nachricht mit derselben Identifikationsnummer und speichert diese, sortiert nach der Sendezeit der „echten“ Nachricht, in diesem Puffer. Sobald ein LP einen Time-Warp durchführt, ermittelt er alle Anti-Nachrichten, deren zugehörige Nachrichten nach (bzgl. Modellzeit) der Rücksprungzeit verschickt wurden, und sendet diese anschließend an die entsprechenden LPs. Dadurch werden alle Nachrichten, die durch inzwischen ungültige Ereignisse erzeugt und verschickt wurden, zurückgezogen.

Eigenschaften des Time-Warp-Verfahrens

Nach der Beschreibung des Time-Warp-Verfahrens ergibt sich die Frage, ob nicht die Gefahr besteht, dass eine optimistisch-parallele Simulation durch ständig wiederkehrende Time-Warps in eine Endlosschleife gerät und dadurch nie die Simulationsendzeit erreicht. Dass dies nie geschieht, lässt sich vergleichsweise einfach begründen.

Bedingt durch das Kausalitätsprinzip, wonach Ereignisse nur durch Ereignisse aus der Vergangenheit erzeugt/beeinflusst werden können, kann jeder LP prinzipiell nur Nachrichten in der Zukunft bezüglich seiner aktuellen Modellzeit erzeugen. Dies gilt auch für alle Nachrichten, deren Empfänger bereits weiter in der Modellzeit vorangeschritten ist, so dass dieser die Nachrichten als Nachzüglernachrichten einstufen und demzufolge einen Time-Warp durchführen muss. Daraus folgt aber auch, dass einerseits der jeweils am weitesten zurückliegende LP niemals eine Nachzüglernachricht erhalten kann, und andererseits, dass nie bei einem Time-Warp eine Rücksprungzeit auftreten kann, die sich in der Vergangenheit dieses langsamsten LPs befindet.

Daraus wiederum folgt, dass es zu jedem Zeitpunkt während einer Simulation eine minimale Modellzeit gibt und diese monoton steigt (auf den Spezialfall der Behandlung gleichzeitiger (bzgl. Modellzeit) Anti-/Nachrichten wird gesondert in Abschnitt 3.6 eingegangen). Da diese monoton steigende, minimale Modellzeit zwangsläufig irgendwann die Simulationsendzeit erreichen muss, ist auch die Terminierung einer optimistisch-parallelen Simulation stets gewährleistet.

In Abbildung 3.8 ist auf der linken Seite das Laufzeitdiagramm eines optimistisch-parallelen Simulationslaufes zu sehen. Analog zum Laufzeitdiagramm eines konservativ-parallelen Simulationslaufes enthält das Diagramm mehrere Graphen, die den Fortschritt in der Modellzeit der einzelnen LPs darstellen. Der wichtigste Unterschied zur konservativ-parallelen Simulation besteht in der Tatsache, dass die Modellzeit der einzelnen LPs bedingt durch Time-Warps auch abnehmen kann. Im abgebildeten Laufzeitdiagramm tritt dies bei zwei LPs jeweils einmal auf.

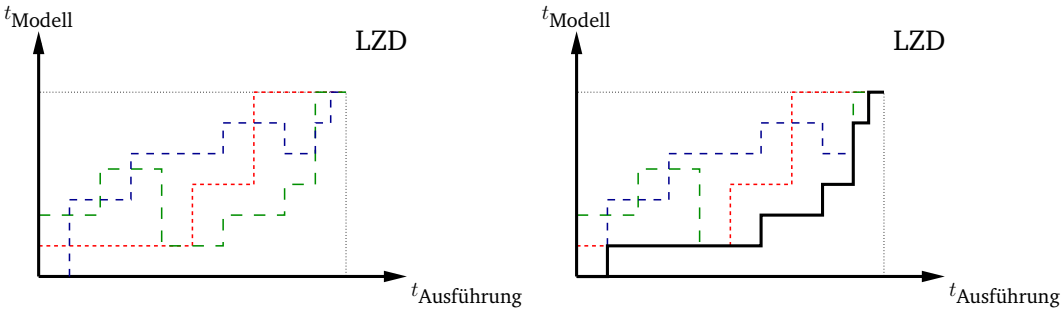

Abbildung 3.8: Laufzeitdiagramme eines optimistisch-parallelen Simulationslaufes (im rechten ist das jeweilige Minimum der Modellzeiten aller LPs hervorgehoben)

Da durch die Möglichkeit des Time-Warps im am weitesten fortgeschrittenen LP auch das Maximum über die aktuellen Modellzeiten aller LPs abnehmen kann, ist dieses im Unterschied zur konservativ-parallelen Simulation nicht mehr als Maß für den Fortschritt eines Simulationslaufes zu gebrauchen. Hingegen lässt sich auch in optimistisch-parallelen Simulationsläufen das Minimum aller Modellzeiten benutzen, da dieses, wie oben beschrieben, in der Modellzeit ausschließlich zunehmen kann. Im zweiten Diagramm von Abbildung 3.8 ist diese minimale Modellzeit, die im folgenden Abschnitt noch ausführlich diskutiert wird, hervorgehoben dargestellt.

Global Virtual Time (GVT)

Da die im vorherigen Abschnitt mehrfach erwähnte minimale Modellzeit aller LPs während eines Simulationslaufes in verschiedenen Bereichen der parallelen Simulation eine zentrale Rolle spielt, hat sie in der Simulationsliteratur eine eigene Bezeichnung erhalten: *global virtual time (GVT)*. Die GVT stellt eine wichtige Grundlage für verschiedene Optimierungsansätze von parallelen Simulationsverfahren dar. Im Folgenden werden einige Informationen zur Berechnung der GVT sowie auf der GVT basierende Optimierungen vorgestellt.

Berechnung der GVT

Die GVT lässt sich zur Laufzeit nicht so leicht berechnen, wie nach ihrer Definition als Minimum der aktuellen Modellzeiten aller LPs zu vermuten ist, es sei denn, man unterbricht den gesamten Simulationslauf für die Dauer der GVT-Berechnung. Diese unter Umständen häufig durchzuführende Zwangssynchronisation würde jedoch den Parallelisierungsgrad der Ausführung deutlich senken und damit den erhofften Geschwindigkeitsvorteil paralleler Simulationen weitgehend eliminieren.

Es gibt in der Simulationsliteratur zahlreiche GVT-Algorithmen, die die GVT iterativ über die LPs ermitteln. Dabei werden in der ersten Iteration die aktuellen Modellzeiten der einzelnen LPs ermittelt und in weiteren Iterationen überprüft, welche

dieser ermittelten Modellzeiten seit der vorherigen Iterationen ungültig geworden sind. Allerdings ist die Korrektheit vieler dieser Algorithmen nie bewiesen worden; von einigen kann man sogar nachweisen, dass sie nicht funktionieren [Meh94].

Weitere Ansätze der GVT-Berechnung bestehen im kontrollierten Steuern bzw. Anhalten einzelner LPs, z. B. in der Form, dass kein LP während einer laufenden GVT-Berechnung einen Time-Warp in einem anderen LP auslösen darf, sondern in diesem Fall mit dem Versand der entsprechenden (Anti-)Nachricht warten muss, bis die GVT-Berechnung beendet ist. Eine Beschreibung von verschiedenen funktionierenden GVT-Algorithmen sowie Listen von Literaturreferenzen auf weitere Verfahren befinden sich sowohl in [Fuj99] als auch in [Meh94].

Speicherfreigabe

Wie bereits bei der Beschreibung des Time-Warp-Verfahrens erwähnt, besteht ein üblicher Weg zur Sicherstellung der Rücksetzmöglichkeit des Simulationsmodellzustandes in der kontinuierlichen Speicherung von Kopien des jeweiligen Zustandes zu jedem Ereigniszeitpunkt. Es ist offensichtlich, dass dadurch der Speicherbedarf eines Simulationsprogrammes während eines Simulationslaufes kontinuierlich zunimmt. Dies stellt insbesondere bei komplexen Simulationsmodellen, deren zu speichernde Zustände ebenfalls groß sind, und/oder vielen Ereignissen, d. h., es müssen viele Kopien angelegt werden, ein ernsthaftes Problem dar.

Einen Ausweg liefert die bei der Vorstellung der GVT gewonnene Erkenntnis, dass ein LP nie einen Time-Warp durchführen kann, dessen Rücksprungzeit vor der GVT liegt. Daraus folgt automatisch, dass alle gespeicherten Simulationsmodellzustände, die zu Modellzeiten vor der GVT gehören, obsolet sind und daher entfernt werden können. Dieser Vorgang der automatischen Entsorgung nicht mehr benötigter gespeicherter Modellzustände wird als *fossil collection* bezeichnet.

Es gibt noch weitere Verfahren zur Verringerung des benötigten Speichers, wie z. B. das künstliche Auslösen von Time-Warps, wodurch auch gültige Modellzustände zur kurzfristigen Speicherfreigabe temporär entfernt werden. Da diese Verfahren jedoch sehr zu Lasten der Simulationsgeschwindigkeit gehen, werden sie nur in speziellen und vor allem gut begründeten Fällen eingesetzt und sollen hier nicht näher betrachtet werden.

Serialisierte Ausgabe

Ein weiteres Problem bei der optimistisch-parallelen Simulation, das sich elegant mit der GVT lösen lässt, ist das Problem der Informationsausgabe bzw. -speicherung während eines Simulationslaufes. Bei sequentiellen und konservativ-parallelen Simulationen kann der Simulationskern bzw. jeder LP jederzeit Informationen über das Simulationsmodell ausgeben und/oder speichern, da durch die garantiert kor-

rekte Abarbeitungsreihenfolge aller Ereignisse nur gültige Ausgaben produziert werden.[2]

Bei optimistisch-parallelen Simulationen hingegen ergibt sich das Problem, dass analog zu den Simulationsmodellzuständen auch erfolgte Ausgaben im Nachhinein ungültig werden. Da sich je nach Ausgabemedium bestimmte Ausgaben jedoch nicht revidieren lassen, besteht ein üblicher Ansatz zur Lösung des Problems darin, alle Ausgaben zunächst inklusive zugehöriger Modellzeit in einem Puffer zu speichern und im Falle der Revision einer Ausgabe diese wieder aus dem Puffer zu entfernen. Erst nach Beendigung eines Simulationslaufes werden alle gespeicherten und damit definitiv gültigen Ausgaben in chronologisch korrekter Reihenfolge ausgegeben. Bei diesem Ansatz ergibt sich aber analog zur Zustandsspeicherung das Problem eines kontinuierlich steigenden Speicherbedarfs zur Simulationslaufzeit. Des Weiteren ist der Verzicht der Ausgabe von Informationen während eines Simulationslaufes problematisch, da unter Umständen bereits zu Beginn eines Simulationslaufes anhand frühzeitiger Ausgaben erkennbar ist, ob das Simulationsmodell fehlerhaft und damit der Rest des potentiell langwierigen Simulationslaufes obsolet ist.

Eine Lösung dieser Teilprobleme liefert analog zur oben beschriebenen Speicherfreigabe die GVT. Da alle Ausgaben, die zu einem Modellzeitpunkt vor der GVT ausgelöst wurden, prinzipbedingt gültig sind, können diese sofort ausgegeben und der durch sie okkupierte Speicher freigegeben werden.

3.6 Weitere Probleme optimistisch-paralleler Simulationen

Neben den in Abschnitt 3.3 diskutierten Problemen gibt es weitere Aspekte, die bei der Implementation von parallelen Simulationen mit verteiltem Simulationsmodell beachtet werden müssen.

Verbot geteilter Referenzen

Eine der größten Einschränkungen für einen Simulationsmodellierer, die mit der Verteilung des Simulationsmodells einhergeht, ist das Verbot der gemeinsamen Referenzierung von Variablen durch verschiedene LPs (genaugenommen durch Simulationsmodellteile, die auf unterschiedlichen LPs ausgeführt werden). Dabei besteht das Problem darin, dass während eines Simulationslaufes jeder LP unabhängig von den anderen in der Modellzeit voranschreitet. Wenn nun zwei LPs Referenzen auf dieselbe Variable besitzen, stellt sich automatisch die Frage, welcher Modellzeit die Belegung dieser Variablen zugehörig ist. Dabei kann insbesondere der kritische Fall auftreten, dass ein LP zu einer bestimmten Modellzeit in eine Variable schreibt und anschließend ein anderer LP zu einer früheren Modellzeit die Variable liest. In diesem Fall wird ein Wert aus der Zukunft in die Vergangenheit übertragen, was einen offensichtlichen Kausalitätsfehler darstellt.

[2] Im Falle der konservativ-parallelen Simulation werden die Ausgaben eventuell in der falschen Reihenfolge ausgegeben. Dies lässt sich aber nachträglich leicht korrigieren.

Aber auch ohne die Übertragung zukünftiger Werte in die Vergangenheit können durch gemeinsam referenzierte Variablen Kausalitätsfehler entstehen, und zwar dann, wenn ein bereits durch einen LP gelesener Wert einer Variablen durch eine Schreiboperation eines anderen zu einem früheren Modellzeitpunkt nachträglich ungültig wird. In optimistisch-parallelen Simulationen kann zusätzlich der Fall eintreten, dass Schreiboperationen im Nachhinein zurückgezogen werden müssen, wenn die schreibenden LPs einen Time-Warp durchführen. Auch in diesem Fall sind alle später (bzgl. Modellzeit) erfolgten Leseoperationen potentiell ungültig.

Die einzige Ausnahme des Verbots sind geteilte Referenzen auf Konstanten. Da diese prinzipbedingt durch die LPs lediglich lesbar sind, können durch sie keine der oben beschriebenen Kausalitätsfehler durch Schreibzugriffe auftreten.[3] Besonders Ressourcen (siehe Abschnitt 2.8) sind vom Verbot geteilter Referenzen betroffen. Da diese jedoch insbesondere für Simulationen mit prozessorientierten Simulationsmodellen unverzichtbar sind, erfolgte im Rahmen dieser Arbeit eine Implementation, die mit dem Verbot geteilter Referenzen kompatibel ist. In Abschnitt 5.4 befindet sich eine Beschreibung dieser Lösung.

Reproduzierbarkeit

Wie bereits in Abschnitt 2.7 diskutiert, besteht eine zentrale Anforderung an Simulationsimplementationen in der Reproduzierbarkeit von Simulationsläufen. Diese Anforderung ist bei parallelen Simulationen aus den gleichen Gründen motiviert wie bei sequentiellen, allerdings ist sie in parallelen Simulationen deutlich schwieriger zu erreichen. Dies liegt unter anderem daran, dass bei verteilten Simulationen neue Situationen hinzukommen, in denen die Reproduzierbarkeit gefährdet ist. Des Weiteren funktionieren die in Abschnitt 2.7 vorgestellten Ansätze zur Reproduzierbarkeitssicherung nicht bei parallelen Simulationen.

Zufallszahlen und -generatoren

Bezüglich der Verwendung von Pseudozufallszahlengeneratoren, die reproduzierbare Folgen von Pseudozufallszahlen generieren, lässt sich festhalten, dass diese in parallelen Simulationen mitnichten das Problem der Reproduzierbarkeit bei nichtdeterministischen Simulationsmodellen lösen. Der Grund dafür liegt darin, dass in parallelen Simulationen die Reihenfolge der Anforderung von Zufallszahlen durch mehrere LPs nicht mehr reproduzierbar ist.

Während sich dieses Problem in konservativ-parallelen Simulationen noch dadurch lösen lässt, dass jeder LP seinen eigenen Zufallsgenerator erhält, tritt bei optimistisch-parallelen Simulationen ein weiteres Problem auf: die wiederholte Anforderung von Zufallszahlen eines LPs nach einem Time-Warp. Damit die Reproduzier-

[3] Dies gilt nur für Konstanten, die bereits zu Simulationslaufbeginn mit ihrem endgültigen Wert belegt werden. Konstanten, die erst später einmalig mit ihrem Wert belegt werden (wie es z. B. das Konstantenkonzept in Java erlaubt), sind hingegen bei geteilter Referenzierung kausalitätsfehleranfällig.

barkeit auch in diesem Fall gewährleistet ist, müssen die Zufallszahlengeneratoren so implementiert werden, dass sie auch bei wiederholten Anfragen erneut dieselben Ergebnisse zurückliefern.

Gleichzeitig eintretende Ereignisse

Wie in Abschnitt 2.7 beschrieben, gefährden Ereignisse mit identischen Eintrittszeitpunkten in sequentiellen Simulationen die Reproduzierbarkeit von Simulationsläufen. Bei parallelen Simulationen sind die potentiellen Folgen gleichzeitiger Ereignisse allerdings deutlich dramatischer: Prinzipiell können sie zu ungültigen und bei optimistisch-parallelen Simulationen zusätzlich auch zu nicht terminierenden Simulationsläufen führen.

Beide Folgen entstehen aus der Tatsache, dass bei parallelen Simulationen mitnichten jede beliebige Serialisierung gleichzeitiger Ereignisse erlaubt ist. Stattdessen lässt sich aus der Forderung nach Kausalität folgern, dass nur diejenigen Abarbeitungsreihenfolgen legitim sind, die die sogenannte *transitive Erzeugungsreihenfolge* respektieren. Darunter ist zu verstehen, dass bei einem kausalen Zusammenhang zweier laut Eintrittszeitpunkt bezüglich Modellzeit gleichzeitiger Ereignisse (d. h., das eine Ereignis wird direkt oder indirekt durch das andere erzeugt), diese in der Reihenfolge ihrer Erzeugung abgearbeitet werden müssen.

In sequentiellen Simulationen ist das Einhalten der transitiven Erzeugungsreihenfolge trivial, da das später erzeugte Ereignis erst dann zu existieren beginnt, wenn das frühere Ereignis gerade abgearbeitet wird. Bei parallelen Simulationen hingegen können beide Ereignisse prinzipiell gleichzeitig und bei optimistisch-parallelen Simulationen nach einem Time-Warp sogar in umgekehrter Reihenfolge abgearbeitet werden.

Alternativ können optimistisch-parallele Simulationen durch Verletzung der transitiven Erzeugungsreihenfolge auch in eine Endlosschleife geraten. Dazu ist lediglich eine Ereigniskette notwendig, in der ein Ereignis indirekt in seinem eigenen LP ein Folgeereignis zur gleichen Modellzeit erzeugt, das in diesem LP einen Time-Warp auslöst, so dass das ursprüngliche Ereignis erneut abgearbeitet wird, wodurch wiederum das Time-Warp-auslösende Ereignis erzeugt wird, usw.

Eine Lösung für alle genannten Probleme liefert erneut eine implizite Priorisierung, die jedoch komplexer ist, als die aus Abschnitt 2.7 für sequentielle Simulationen. Die Grundidee besteht darin, den Datentyp der Modellzeit mit zusätzlichen Attributen so anzureichern, dass sich je zwei Modellzeiten stets in mindestens einem Attribut unterscheiden und so bei Gleichzeitigkeit der ursprünglichen Modellzeit anhand der Attribute stets eine eindeutige Sortierung möglich ist.

Im Folgenden wird ein Modellzeitdatentyp beschrieben, der alle genannten Eigenschaften aufweist und durch den gleichzeitig Ereignisse automatisch unter Einhaltung der transitiven Erzeugungsreihenfolge sortiert werden können. Zur besseren Unterscheidung wird die bisher verwendete Modellzeit *skalare* und die neue *strukturierte* Modellzeit genannt.

Die strukturierte Modellzeit besteht aus vier Attributen. Dabei handelt es sich im Einzelnen um:

- Die bisher verwendete, skalare Modellzeit. Diese geht unverändert als erstes Attribut in Form eines einfachen Zahlentyps in die strukturierte Modellzeit ein.
- Das Altersattribut, bei dem es sich ebenfalls um einen einfachen Zahlentyp handelt und dessen Wert folgendermaßen bestimmt wird: Alle initialen Ereignisse sowie alle Ereignisse, deren Erzeugerereignisse eine unterschiedliche skalare Modellzeit aufweisen, erhalten als Alter den Wert null. Wird hingegen ein Ereignis von einem gleichzeitigen Ereignis (bzgl. skalarer Modellzeit) erzeugt, so erhält dieses neue Ereignis zwar die gleiche skalare Modellzeit, dafür erhält jedoch das Altersattribut den um eins inkrementierten Alterswert des erzeugenden Ereignisses.
- Der LP, der das Ereignis mit dieser strukturierten Modellzeit generiert hat. Dieses Attribut kann durch beliebige Datentypen dargestellt werden, solange sich die Elemente dieses Datentyps eindeutig sortieren lassen. Eine übliche Implementationsvariante besteht in der eindeutigen Nummerierung aller LPs und der Darstellung des LP-Attributes als einfachen Zahlentyp.
- Die Anzahl der Ereignisse, die der LP bereits generiert hat, inklusive dem, das diese strukturierte Modellzeit erhalten soll. Bei diesem Attribut handelt es sich ebenfalls wieder um einen einfachen Zahlentyp.

Ein Vergleich zweier derartiger strukturierter Modellzeiten erfolgt durch einen attributweisen Vergleich, wobei die Attribute in der obigen Reihenfolge durchgegangen werden und die erste auftretende Ungleichheit den Ausschlag gibt. Es ist hervorzuheben, dass durch den beschriebenen Aufbau eine Gleichheit zweier Modellzeiten nur dann auftreten kann, wenn sie den Eintrittszeitpunkt desselben Ereignisses beschreiben.

Wie in [Fuj99] und [Meh94] gezeigt wird, garantiert die beschriebene strukturierte Modellzeit bei der Simulation deterministischer Simulationsmodelle sowohl die Einhaltung der transitiven Erzeugungsreihenfolge als auch die Reproduzierbarkeit der Simulationsläufe.[4]

[4] Sowohl in [Fuj99] als auch in [Meh94] werden das dritte und vierte Attribut zu einem gemeinsamen Attribut in Form eines strukturierten Datentyps namens *id* zusammengefasst. Infolgedessen wird die strukturierte Modellzeit in diesen Werken als Tripel (klassische Modellzeit, Alter, id) betrachtet.

KAPITEL 4

REALISIERUNG VON OPTIMISTISCH-PARALLELEN PROZESSEN

Nachdem in den vorangegangenen Kapiteln die theoretischen Grundlagen sowie die prinzipiellen Implementationsmöglichkeiten von sequentiellen und parallelen Simulationen betrachtet wurden, soll sich dieses Kapitel nun mit den Fragestellungen bezüglich einer konkreten Umsetzung von Prozessen in optimistisch-parallelen Simulationsimplementationen beschäftigen.

Zunächst werden die prinzipiellen Möglichkeiten der Einbettung von Prozessen in optimistisch-parallelen Simulationskernen diskutiert. Anschließend werden die grundsätzlichen Anforderungen an eine Prozessimplementation ermittelt. Dabei werden die programmiersprachlichen Konzepte Coroutine und Continuation als elementare Implementationsgrundlage beschrieben and klassifiziert.

In den darauffolgenden Abschnitten werden programmiersprachenunabhängig verschiedene Implementationsansätze für Coroutinen/Continuations vorgestellt und auf ihre prinzipielle Anwendbarkeit bezüglich des Implementationsziels dieser Arbeit analysiert. Den Abschluss des Kapitels bildet die detaillierte Beschreibung des konkret im Rahmen dieser Arbeit verwendeten Implementationsansatzes.

4.1 Prozesse in optimisch-parallelen Simulationskernen

Bevor man sich mit der Implementation von optimisch-parallelen Prozessen beschäftigt, stellt sich zunächst die Frage, wie das Prozesskonzept prinzipiell in einem optimisch-parallelen Simulationskern eingebettet werden kann. Insbesondere stellt sich bei näherer Betrachtung dieses Problems die Frage, wie die in Abschnitt 2.8 vorge-

stellte, für eine intuitive Modellierung propagierte prozessorientierte Modellierungssicht mit der in Abschnitt 3.3 beschriebenen Partitionierung des Simulationsmodells in Einklang zu bringen ist.

Die operative Grundlage eines optimisch-parallelen Simulationskerns stellen die LPs dar, die man, wie ebenfalls in Abschnitt 3.3 beschrieben wurde, als einzelne, sequentielle Simulationskerne betrachten kann. In diesem Sinne bedeutet eine Partitionierung des Simulationsmodells also zunächst lediglich die Verteilung der vorhandenen Prozesse auf die verfügbaren LPs, die dann während eines Simulationslaufes ausschließlich die ihnen zugewiesenen Prozesse ausführen. Dies ist auch die Sicht, die dem Simulationsmodellierer präsentiert werden sollte, da ihm so zwar eine grobe Steuerung der Parallelisierung möglich ist, während ihm gleichzeitig die internen Arbeitsabläufe des optimisch-parallelen Simulationskerns verborgen bleiben.

Damit eine solche Abstraktion gelingt, müssen der Simulationskern bzw. seine LPs einige zusätzliche Aufgaben übernehmen. Zunächst müssen die Prozessaktivierungen in entsprechende Nachrichten zwischen den LPs übersetzt werden. Dabei ist beim Versand der Nachrichten darauf zu achten, dass der jeweils empfangende LP auch derjenige ist, der dem der zu aktivierenden Prozess zugewiesen wurde, der also den Prozess ausführt. Des Weiteren müssen in Hinblick auf einen möglichen Time-Warp die Prozesszustände regelmäßig gespeichert werden. Auch diese Aufgabe wird von den LPs für die jeweils von ihnen bearbeiteten Prozesse übernommen. Eng damit verbunden, und ebenfalls Aufgabe der LPs, ist die Wiederherstellung eines früheren Prozesszustandes im Falle eines Time-Warps. Die Erkennung, ob eine Prozessaktivierungen Time-Warp auslösen, das Zurücksetzen von LPs und das Versenden von Anti-Nachrichten erfordern hingegen bei der Unterstützung prozessorientierter Simulationsmodelle keinen zusätzlichen Aufwand gegenüber „klassischen“, auf dem reinen Nachrichtenaustausch basierenden optimistisch-parallelen Simulationskernen.

Ein Problem, das sich bei der Partitionierung eines prozessorientierten Simulationsmodells nicht einfach lösen lässt, ist das in Abschnitt 3.6 beschriebene Verbot gemeinsamer Referenzen durch Simulationsmodellteile auf verschiedenen LPs. Dies ist darin begründet, dass während einer optimisch-parallelen Simulationsausführung jeder Simulationsmodellteil zu jedem Ausführungszeitpunkt eine eigene, eindeutige aktuelle Modellzeit besitzen muss. Da dies bei einer von mehreren Prozessen verschiedener LPs gemeinsam referenzierten Komponente nicht der Fall wäre, bedeutet dies, dass nur solche prozessorientierten Simulationsmodelle partitioniert werden können, die sich von vornherein dieser Einschränkung unterwerfen. Es ist allerdings möglich, die Situation zumindest teilweise zu entschärfen, indem man dem Simulationsmodellierer Ersatzkonzepte mit ähnlicher Funktionalität zur Verfügung stellt, wie es in dieser Arbeit mit der in Abschnitt 5.4 vorgestellten optimisch-parallelen Ressourcenimplementation praktiziert wurde.

4.2 Grundlagen der Prozessimplementation

Wie bereits in Abschnitt 2.8 beschrieben wurde, ist die Implementation eines ereignisgesteuerten Simulationskerns weitgehend unabhängig von der verwendeten Modellierungssicht. Der wesentliche strukturelle Unterschied besteht in der Art von Ereignissen, die im Ereigniskalender eingetragen werden können. Während es sich bei ereignisorientierten Simulationsmodellen direkt um die Ereignisse handelt, die im Rahmen der Simulationsmodellbildung identifiziert wurden, werden bei prozessorientierten Simulationsmodellen Prozessaktivierungen als Ereignisse eingetragen. Damit einher geht eine unterschiedliche Behandlung in jeder Iteration der zentralen Simulatorschleife.

Bei „echten“ Ereignissen wird die zugehörige Ereignisroutine direkt aufgerufen und nach deren Beendigung im Simulationskern fortgefahren. Dabei stellt die Ereignisroutine selbst aus Implementationssicht nichts weiter als eine Methode ohne Rückgabewert dar. Bei Prozessaktivierungen hingegen muss der zugehörige Prozesslebenslauf gestartet bzw. fortgesetzt und anschließend dessen Unterbrechung bzw. Beendigung abgewartet werden. Ein solches Verhalten lässt sich jedoch nicht durch Verwendung einfacher Methoden ausdrücken. Wie bereits in Abschnitt 2.8 beschrieben, werden zwei zusätzliche Funktionalitäten benötigt, die das Unterbrechen (*suspend*) und Fortsetzen (*continue*) inklusive Speicherung bzw. Wiederherstellung des Prozesszustandes übernehmen.

Eigenschaften einer idealen Prozessimplementation

Perumalla und Fujimoto haben in [PF98] fünf, voneinander unabhängige Eigenschaften vorgestellt, die ein ideales, Prozesse unterstützendes Simulationssystem erfüllen bzw. unterstützen muss:

1. Methoden können lokale Variablen deklarieren und benutzen,
2. Methodenaufrufe können geschachtelt erfolgen,
3. Methodenaufrufe können rekursiv und reentrant erfolgen,
4. Aufrufe zur Modellzeitfortführung können innerhalb jeder Methode auftreten (d. h. nicht nur in der Lebenszyklusmethode, sondern auch in von dieser aufgerufenen Methoden),
5. Aufrufe zur Modellzeitfortführung können überall auftreten, wo auch andere Anweisungen auftreten können (d. h. insbesondere auch innerhalb von Verzweigungen, Schleifen, . . .).

Die ersten drei Eigenschaften sind Anforderungen an die Simulationssprache, die vom Simulationssystem zur Formulierung von Simulationsmodellen zur Verfügung gestellt wird. Wie bereits in Abschnitt 2.2 beschrieben, werden in dieser Arbeit ausschließlich Simulationssysteme in Form von Simulationsbibliotheken betrachtet. Da bei diesen die jeweilige Simulationssprache mit der dem Simulationssystem zugrundeliegenden Programmiersprache identisch ist, müssen die ersten drei Eigenschaften demzufolge zunächst von eben dieser Programmiersprache unterstützt werden,

was bei praktisch allen modernen, prozeduralen Programmiersprachen der Fall ist. Des Weiteren muss jedoch sichergestellt werden, dass eine konkrete Prozessimplementation nicht bereits durch die Programmiersprache zugesicherte Eigenschaften nachträglich wieder einschränkt.

Die beiden letztgenannten Eigenschaften stellen hingegen unabhängig von der benutzten Simulationssprache eine Herausforderung dar. Bei den Aufrufen zur Modellzeitfortführung handelt es sich um alle Aufrufe, die einen Prozess unterbrechen und damit dem Simulationskern erlauben, in der Modellzeit voranzuschreiten.

Es existieren sehr wohl Simulationssysteme, die nicht alle fünf genannten Eigenschaften erfüllen. Allerdings führt jede nicht unterstützte Eigenschaft automatisch zu einer Einschränkung für den Simulationsmodellierer während der Erstellung des Simulationsmodells. Daher ist es ein Ziel dieser Arbeit, eine Prozessimplementation zur Verfügung zu stellen, die alle genannten Anforderungen erfüllt.

Im Folgenden werden verschiedene Implementationsansätze und die ihnen zugrundeliegenden abstrakten Konzepte vorgestellt. Dabei wird zu jedem Ansatz diskutiert, welche der fünf Eigenschaften durch ihn erfüllt bzw. verletzt werden und wie er sich für die Implementation von Prozessen in einer optimistisch-parallelen Simulationsbibliothek eignet.

4.3 Coroutinen

Zur Implementation von Prozessen wird ein abstraktes Konzept benötigt, das als *Coroutine* bezeichnet wird. Allerdings sind für dieses Konzept bis heute mehrere, teilweise widersprüchliche Definitionen in Gebrauch. Allen diesen Definitionen sind jedoch die beiden folgenden wesentlichen Eigenschaften einer Coroutine gemein (zitiert nach [Mar80]):

- die Abarbeitung einer Coroutine kann durch Abgabe der Steuerung unterbrochen/deaktiviert werden; sobald die Coroutine wieder aktiv wird, setzt sie ihre Arbeit genau an der Stelle fort, wo sie unterbrochen wurde, und
- die Werte aller lokalen Daten einer Coroutine bleiben zwischen zwei aufeinanderfolgenden Aufrufen erhalten.

Diese beiden Eigenschaften stellen exakt die Anforderungen dar, die zur Implementation von Prozessen benötigt werden.

Im Gegensatz zum Subroutinen-Konzept, das heutzutage in allen gängigen prozeduralen Programmiersprachen (z. B. in *Pascal* als *Prozedur*, in *C* als *Funktion*, in *Java* als *Methode*) verfügbar ist, bieten nur wenige Programmiersprachen eine native Unterstützung von Coroutinen an. Unter diesen Ausnahmen befinden sich vor allem ältere Programmiersprachen wie Simula [DN66], BETA [MPN93], Modula-2 [Wir83] und Icon [GG96]. Dennoch lassen sich, wie auch in den folgenden Abschnitten noch gezeigt wird, Coroutinen sehr wohl in anderen Programmiersprachen realisieren.

Um der Vielzahl von unterschiedlichen Coroutinen-Implementationen zu begegnen, wurden in [MI04] mehrere Klassifikationen, basierend auf grundlegenden Ei-

genschaften der jeweils implementierten Coroutinen, eingeführt. Im Folgenden werden diese Klassifikationen vorgestellt und die jeweiligen Klassen von Coroutinen-Implementationen hinsichtlich der fünf in Abschnitt 4.2 genannten Eigenschaften bewertet.

Klassifikation nach Symmetrie

Man kann Coroutinen nach der Art, wie die Übergabe der Steuerung bei einer Unterbrechung oder Fortsetzung erfolgt, in *symmetrische* und *asymmetrische* Coroutinen unterteilen.

Bei symmetrischen Coroutinen erfolgt die Steuerungsweitergabe direkt von einer Coroutine zu einer anderen. Umgesetzt wird dieser Steuerungswechsel in der Regel durch den Aufruf einer speziellen Methode (*transfer*)[1]. Dabei erfolgt dieser Wechsel, ohne dass die betroffenen Coroutinen dadurch eine Aufrufer-Aufgerufener-Beziehung eingehen, d. h. insbesondere, dass die aktivierte Coroutine nach ihrer Beendigung/Unterbrechung die Steuerung nicht automatisch der vorher aktiven Coroutine zurückgibt. Stattdessen ist in aus symmetrischen Coroutinen zusammengesetzten Programmen stets die aktive Coroutine dafür zuständig, die nachfolgend aktive Coroutine zu bestimmen und dann die Steuerung an diese zu übertragen. Insofern erklärt sich auch die Bezeichnung *symmetrische* Coroutine: Der zum Betreten einer symmetrischen Coroutine benötigte Mechanismus ist der gleiche, der zum Verlassen derselbigen (und gleichzeitig zum Betreten der nächsten) verwendet wird.

Durch die fehlende Aufrufhierarchie verliert auch das Hauptprogramm seine Rolle als solches. Von Conway, dem die erstmalige Erwähnung des Begriffes *Coroutine* zugeschrieben wird, stammt die folgende Beschreibung [Con63]:

> "Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program."

In Quelltext 4.1 ist eine Implementation des klassischen Erzeuger-Verbraucher-Problems durch zwei entsprechende symmetrische Coroutinen, notiert in Pseudocode, zu sehen. Ein derartiges gegenseitiges Aufrufen würde bei der Benutzung von herkömmlichen Subroutinen nach kurzer Zeit zu einem Überlaufen des Programmstacks und damit zum Abbruch bzw. Absturz des Programmes führen.

Bei asymmetrischen Coroutinen-Implementationen hingegen bleibt die von Subroutinen bekannte Aufrufhierarchie erhalten. Asymmetrische Coroutinen werden zunächst wie normale Subroutinen aufgerufen, können sich dann aber durch den Aufruf einer speziellen Methode (*suspend*) selbst unterbrechen. Dabei wird die Steuerung automatisch wieder der ursprünglich rufenden Routine übertragen. Die Coroutine kann anschließend jederzeit mittels einer zweiten speziellen Methode (*continue*) fortgesetzt werden, wobei auch bei einer erneuten Unterbrechung die

[1] Der Name der Methode ist frei wählbar. Die Verwendung des genannten und der noch folgenden konkreten Namen erfolgt lediglich zur besseren Verständlichkeit der weiteren Ausführungen.

```
1  coroutine producer() {
2      while (true) {
3          ... // produce something
4          transfer (consumer)
5      }
6  }
7  coroutine consumer() {
8      while (true) {
9          ... // consume something
10         transfer (producer)
11     }
12 }
```

Quelltext 4.1: Pseudocode-Beispiel für symmetrische Coroutinen

```
1  coroutine counter() {
2      int number = 0
3      while (true) {
4          number++
5          print (number)
6          suspend()
7      }
8  }
```

Quelltext 4.2: Pseudocode-Beispiel für asymmetrische Coroutinen

Steuerung zwangsläufig auf die Routine übergeht, die diese Fortsetzung verursachte. Eine Programmierweise, bei der die Steuerung von Coroutine zu Coroutine weitergereicht wird, ist bei asymmetrischen Coroutinen nur über Umwege möglich.

Asymmetrische Coroutinen werden auch als *semi-symmetrische* Coroutinen oder als *Semi-Coroutinen* bezeichnet [DDH72]. In Quelltext 4.2 ist eine sehr einfache Anwendung einer asymmetrischen Coroutine zu sehen. Diese gibt bei jeder Fortsetzung die Anzahl ihrer bisherigen Fortsetzungen aus.

Es sollte festgehalten werden, dass symmetrische und asymmetrische Coroutinen die gleiche Ausdrucksstärke besitzen, d. h., jedes Problem, dass sich mit symmetrischen Coroutinen lösen lässt, ist auch mit asymmetrischen Coroutinen lösbar und umgekehrt [MI04]. Dennoch sollte man im Sinne lesbarer Programme auf diejenige Coroutinenklasse zurückgreifen, in der sich das jeweilige Problem eleganter bzw. intuitiver darstellen lässt.

Für den Anwendungsfall dieser Arbeit, die Implementation von Prozessen, sind asymmetrische Coroutinen eher geeignet. Die Begründung liegt in der Arbeitsweise eines Simulationsprogrammes (siehe Abschnitt 2.8), die eine feste Aufrufhierarchie zwischen der Hauptmethode des Simulationskerns und den von diesem gerufenen Prozessroutinen vorsieht: Prozesslebensläufe werden stets von der Hauptmethode des Simulationskerns aufgerufen und sollen danach stets die Steuerung an diesen zurückgeben; eine Steuerungsweitergabe direkt von einem Prozess zu einem ande-

ren ist nicht vorgesehen. Dennoch existieren einzelne Simulationsbibliotheken, deren Prozessimplementationen auf symmetrischen Coroutinen basieren (z. B. ODEM [Fis82] bzw. dessen Nachfolger ODEMx [Ger03] oder JAVASIMULATION [Hel00]). In diesen Bibliotheken rufen sich die Prozesse gegenseitig auf, indem jeder Prozess unmittelbar vor seiner Beendigung/Unterbrechung den als nächstes zu aktivierenden Prozess ermittelt und dann die Steuerung zu diesem überträgt. Der Simulationskern wird dabei im Wesentlichen auf die Datenstruktur des Ereigniskalenders reduziert, mittels dessen die Prozesse ihre Nachfolger bestimmen.

Klassifikation nach Beschränkungen

Eine weitere Klassifikation bezieht sich darauf, inwieweit eine Coroutinen-Implementation mit Einschränkungen für den Coroutinenentwickler und/oder -anwender verbunden ist. Je nachdem, ob derartige Einschränkungen existieren, werden Coroutinen in *beschränkte* und *unbeschränkte* Coroutinen (nach [MI04] *constrained* und *first-class coroutines*) unterteilt.

Einschränkungen können z. B. darin bestehen, dass externe Coroutinenaufrufe nur von bestimmten Stellen aus erfolgen dürfen, wie z. B. bestimmte Iteratorimplementationen, die nur innerhalb von Schleifen erlaubt sind. Weitere mögliche Einschränkungen bestehen in der Begrenzung der zur Verfügung stehenden Sprachmittel innerhalb einer Coroutine.

Grundsätzlich gilt, dass eine Prozessimplementation, die auf eingeschränkten Coroutinen basiert, ebenfalls diesen Einschränkungen unterworfen ist. Da dadurch prinzipiell alle in Abschnitt 4.2 genannten Bedingungen gefährdet sind, werden in der weiteren Arbeit ausschließlich Implementationsansätze für unbeschränkte Coroutinen verfolgt.

Klassifikation nach Erhaltung des Programmstacks

Als stackerhaltend werden Coroutinen-Implementationen bezeichnet, bei denen die Aufrufe zur Unterbrechung (je nach Symmetrie *suspend* oder *transfer*) nicht nur in der Coroutine selbst, sondern auch in einer von der Coroutine aufgerufenen Subroutine bzw. in noch tiefer verschachtelten Unterroutinen erfolgen darf. Bei einer Fortsetzung der Coroutine wird erwartet, dass die gesamte Aufrufhierarchie zum Zeitpunkt der Unterbrechung wiederhergestellt wird.

Übertragen in die Simulationswelt würde die Benutzung einer nicht stackerhaltenden Coroutinen-Implementation zur ausschließlichen Verwendung von Prozessen führen, die sich nur direkt innerhalb ihrer Lebenszyklusmethode unterbrechen können. Bezogen auf die Anforderungen von Perumalla und Fujimoto in Abschnitt 4.2 bedeutet dies eine automatische Verletzung des vierten Punktes. Da dieser Umstand wiederum mit einer starken Einschränkung bei der Simulationsmodellerstellung einhergeht, werden in der weiteren Arbeit ausschließlich stackerhaltende Coroutinen-Implementationen als Kandidaten für eine Prozessimplementation akzeptiert.

4.4 Continuations

Ein mit Coroutinen eng verwandtes, abstraktes Konzept, das ebenfalls bei der Implementation von Prozessen eine wichtige Rolle spielt, sind Continuations. Unter einer *Continuation* versteht man eine gespeicherte Form des Laufzeitzustandes eines Programmes bzw. eines Programmteils. Dabei enthält eine Continuation alle notwendigen Informationen, um eine spätere Fortsetzung des Programmes von dem gespeicherten Zustand aus zu gewährleisten.

Eine alternative und eher theoretische, aber äquivalente Beschreibung von Continuations ist die Folgende, die Continuations als Funktionen auffasst, die den Rest einer Programmausführung berechnen:

> "In theory a continuation is a function that computes 'the rest of the program', or 'its future'." [Pet99]

Die Äquivalenz beider Beschreibungen ergibt sich aus der Tatsache, dass ein gespeicherter Laufzeitzustand das Verhalten des Programmes in der Zukunft vollständig festlegt, d. h., von demselben gespeicherten Laufzeitzustand fortgesetzte Programme werden sich bei gleichen Eingaben immer identisch verhalten.

Für diese Arbeit ist die Betrachtung von Continuations als gespeicherte bzw. zu speichernde Laufzeitzustände geeigneter, da die praktische Umsetzung einer derartigen Laufzeitzustandsspeicherung in prozeduralen Sprachen das Kernproblem bei den späteren Ausführungen darstellen wird.

Coroutinen und Continuations

Prinzipiell stellen Coroutinen und Continuations zwei völlig eigenständige Konzepte dar: Coroutinen sind über ihr Verhalten zur Laufzeit definiert, Continuations allein als eine Form der Speicherung von Laufzeitzuständen. Sowohl aus Anwender- als auch aus Entwicklersicht erscheinen jedoch Coroutinen und Continuations eher wie zwei verschiedene Teilaspekte desselben abstrakten Konzeptes.

Zunächst kann festhalten werden, dass jede Continuation-Implementation automatisch ein Coroutinenverhalten umsetzt: Bei der Fortsetzung einer Continuation wird exakt dort fortgesetzt, wo zum Zeitpunkt der Erstellung der Continuation unterbrochen wurde; der bei der Unterbrechung gültige Kontext bleibt dabei als Bestandteil der Continuation erhalten.

Gleichzeitig enthält aber auch jede Coroutinen-Implementation die Implementation einer Continuation. Der bei einer Coroutinen-Implementation zwangsläufig zu speichernde Kontext stellt zusammen mit der ebenfalls zu sichernden Information, an welcher Stelle sich die Coroutine im Moment der Unterbrechung befunden hat, nicht anderes als eine Continuation dar. Zusammenfassend kann also festgehalten werden:

> Jede Implementation einer Coroutine oder Continuation ist bzw. enthält automatisch die Implementation des jeweils anderen Konzeptes; es stellt

sich lediglich die Frage, welchen Klassen die jeweilige Coroutinen- bzw. Continuation-Implementation angehört.

Klassifikation von Continuations

Die letzte Aussage im vorherigen Abschnitt nimmt bereits Bezug auf die Klassifikationen von Continuations, die in den folgenden Teilabschnitten vorgestellt werden. Dabei wird, analog zur Vorstellung der Klassifikationen von Coroutinen-Implementationen, bei jeder vorgestellten Klassifikation analysiert, wie sich die jeweils betrachtete Klasse von Continuation-Implementationen zur Implementation von optimistisch-parallelen Prozessen eignen.

Klassifikation nach Umfang des gespeicherten Zustands

Eine erste Klassifikation von Continuation-Implementationen ist die Unterteilung in *globale* und *lokale* Continuations, letztere werden mitunter auch als *partielle* Continuations [MI04] oder als *Subcontinuations* [HDA94] bezeichnet. Eine globale Continuation-Implementation entspricht exakt der oben beschriebenen Funktion, die den Rest des gesamten Programmes berechnet. In einer entsprechenden Implementation wird demzufolge auch bei jeder Unterbrechung einer Continuation der Laufzeitzustand des gesamten Programmes gesichert.

Während globale Continuations vor allem in funktionalen Programmiersprachen häufig anzutreffen sind, ist ihre Benutzung in prozeduralen Programmiersprachen eher unüblich. Der Grund dafür liegt vor allem in der komplizierten Verwendung, die sich daraus ergibt, dass bei jeder Fortsetzung einer globalen Continuation die gesamte bis dahin gültige Aufrufhierarchie zerstört und durch die gespeicherte ersetzt wird.

Bei lokalen Continuations wird hingegen die Menge der zu speichernden Informationen dahingehend eingeschränkt, dass nur bestimmte Programmteile (in prozeduralen Programmiersprachen in der Regel Subroutinen) unterbrochen und später wieder fortgesetzt werden können.

Bezüglich der Verwandtschaft von Coroutinen- und Continuation-Implementationen lässt sich festhalten, dass symmetrische Coroutinen üblicherweise mit globalen Continuation-Implementationen einhergehen, während asymmetrische Coroutinen in der Regel zusammen mit lokalen Continuation-Implementationen auftreten [MI04]. Die verbleibenden, anderen beiden Kombinationen sind prinzipiell aber auch möglich.

Aufgrund der erleichterten Handhabung und der Tatsache, dass sich bei der Betrachtung von Coroutinenklassifikationen asymmetrische Coroutinen als die günstigere Wahl erwiesen haben, erscheinen lokale Continuations geeigneter für die Implementation von Prozessen.

Klassifikation nach Möglichkeit der mehrfachen Fortsetzung

Die Erkenntnis, dass in vielen Anwendungsfällen von Continuations selbige maximal einmal zur Fortsetzung benutzt werden, führte zur Definition der Klasse der *einfachfortsetzbaren Continuations* (*One-Shot-Continuations*). Continuations hingegen, die beliebig oft fortgesetzt werden können, werden als *mehrfachfortsetzbare Continuations* (*Multi-Shot-Continuations*) bezeichnet.

In [MI04] wird die Praxisrelevanz von mehrfachfortsetzbaren Continuations nahezu vollständig bezweifelt:

> "Although conventional first-class continuation mechanisms allow a continuation to be invoked multiple times, in virtually all their useful applications are invoked only once."

Es ist dementgegen festzuhalten, dass mehrfachfortsetzbare Continuations für die Implementation von optimistisch-parallelen Simulationsprozessen zwingend benötigt werden. Eine Prozessimplementation, die auf einfachfortsetzbaren Continuations basiert, würde im fertigen Simulationsprogramm jegliche Time-Warps verbieten, da die einzige mögliche Fortsetzung der Continuation bereits bei der regulären Fortsetzung des Prozesses erfolgt ist.

4.5 Zusammenfassung der ermittelten Anforderungen

In Abschnitt 4.3 wurden während der Vorstellung der verschiedenen Klassifikationsmöglichkeiten von Coroutinen-Implementation die grundlegenden Anforderungen ermittelt, die eine Implementation für die Umsetzung von optimistisch-parallelen Simulationsprozessen erfüllen muss:

- Asymmetrie: die implementierten Coroutinen sollen sich bezüglich der aufrufenden Routine wie eine normale Subroutine verhalten,
- Unbeschränktheit: die konkreten Coroutinen sollen bezüglich ihrer Verwendung keinen Einschränkungen unterworfen sein und
- Stackerhaltung: Aufrufe zur Unterbrechung einer Coroutine sollen nicht nur im Quelltext derselbigen, sondern auch in von der Coroutine aufgerufenen Subroutinen erfolgen dürfen.

Zusätzlich wurden in Abschnitt 4.4 die wesentlichen Anforderungen identifiziert, welche die gleichzeitig erfolgte Continuation-Implementation erfüllen muss:

- Lokalität: eine Continuation soll mitnichten den Laufzeitzustand eines gesamten Programmes, sondern nur den aktuellen Zustand jeweils einer Coroutine umfassen und
- Mehrfachfortsetzbarkeit: eine einmal gespeicherte Continuation soll beliebig oft fortgesetzt werden können.

4.6 Implementation von Coroutinen/Continuations

Aus den beiden, zu Beginn von Abschnitt 4.3 beschriebenen, fundamentalen Eigenschaften von Coroutinen ergeben sich zwei verschiedene Arten von Informationen, die bei der Unterbrechung einer Coroutine gesichert werden müssen:

- Informationen bezüglich des *Ablaufes*: bei der Unterbrechung einer Coroutine müssen alle notwendigen Informationen gespeichert werden, die eine spätere Fortsetzung an derselben Stelle des Programmes ermöglichen (*Ablaufsteuerung*) und
- Informationen bezüglich des *Kontextes*: die Werte aller lokalen Variablen einer Coroutine müssen bei jeder Unterbrechung selbiger gesichert und bei einer Fortsetzung wiederhergestellt werden (*Kontextsicherung*).

Prinzipiell münden sowohl die Implementation der Ablaufsteuerung als auch die der Kontextsicherung in der Speicherung der jeweiligen Informationen in ein und derselben Continuation. Da sich jedoch die Ermittlung der konkreten Daten in beiden Fällen stark unterscheidet und des Weiteren die verschiedenen Verfahren beider Arten weitgehend frei kombinieren lassen, werden in den folgenden Teilabschnitten die Implementationsansätze separat nach Verfahren zur Ablaufsteuerung und Kontextsicherung vorgestellt.

Implementation der Ablaufsteuerung

Ablaufsteuerung in Maschinensprache/Assembler

Knuth hat in [Knu68] einen Ansatz zur Implementation symmetrischer Coroutinen durch Vereinfachung des herkömmlichen Subroutinenaufrufes beschrieben. In diesem Ansatz werden die der Coroutine zu übergebenden Parameter an eine vordefinierte Stelle im Speicher abgelegt und anschließend wird direkt in die Coroutine gesprungen. Diese speichert umgehend den vorherigen Absprungpunkt, so dass die ursprüngliche Co- oder Subroutine später wieder fortsetzend angesprungen werden kann.

Dieser Ansatz besticht durch seine Einfachheit, lässt sich in dieser Form allerdings nur in maschinennahen Sprachen bzw. Assembler umsetzen, da sich in höheren Programmiersprachen der dort bereits existierende Subroutinenaufruf nicht nachträglich vereinfachen lässt [Tat00].

Ablaufsteuerung durch unbedingte Sprünge

Ein bezüglich des Programmierstils nur unwesentlich abstrakterer Implementationsansatz ist die Realisierung der Ablaufsteuerung durch unbedingte Sprünge (*goto*) und globale Variablen. Die Grundidee liegt in der Implementation von Coroutinen als Subroutinen, die bei jeder Unterbrechung eine Information über den Unterbrechungsort in einer globalen Variablen hinterlegen. Bei einer Fortsetzung springen

```
1  int function(void) {
2      static int i, state = 0;
3      switch (state) {
4          case 0: goto LABEL0;
5          case 1: goto LABEL1;
6      }
7      LABEL0: /* start of function */
8      for (i = 0; i < 10; i++) {
9          state = 1; /* so we will come back to LABEL1 */
10         return i;
11         LABEL1:; /* resume control straight after the return */
12     }
13 }
```

Quelltext 4.3: Beispiel für eine Coroutinen-Implementation durch unbedingte Sprünge in der Programmiersprache C (entnommen aus [Tat00])

```
1  int function(void) {
2      static int i, state = 0;
3      switch (state) {
4          case 0: /* start of function */
5          for (i = 0; i < 10; i++) {
6              state = 1; /* so we will come back to "case 1" */
7              return i;
8              case 1:; /* resume control straight after the return */
9          }
10     }
11 }
```

Quelltext 4.4: Beispielumsetzung der Grundidee der Coroutinen-Implementation von Tatham [Tat00]

diese Subroutinen dann, basierend auf den gespeicherten Werten, die der Unterbrechungsstelle folgende Anweisung an und realisieren dadurch das Coroutinenverhalten. Eine Beispielumsetzung in der Programmiersprache C ist in Quelltext 4.3 zu sehen.

Dieser Ansatz eignet sich ausschließlich zur Umsetzung asymmetrischer Coroutinen. Die parallel erfolgte Continuation-Implementation ist lokal und einfachfortsetzbar. Des Weiteren besitzt der Ansatz zwei prinzipielle Schwächen: Zum einen ist er nur in Programmiersprachen umsetzbar, die über eine unbedingte Sprunganweisung verfügen. Zum anderen lassen sich mit diesem Ansatz stackerhaltende Coroutinen nicht bzw. nur mit nicht vertretbarem Aufwand implementieren.

Auch die in der Literatur häufig referenzierte Implementation von (asymmetrischen) Coroutinen in C von Tatham [Tat00] basiert auf der Umsetzung der Ablaufsteuerung durch unbedingte Sprünge. Allerdings kann Tatham die direkte Nutzung von Sprunganweisungen dadurch umgehen, dass er eine Eigenschaft der Programmiersprache C ausnutzt: In Mehrfachverzweigungen (*switch-case*) sind die zu-

gehörigen *case*-Anweisungen innerhalb des zugehörigen *switch*-Blockes an keine Blockstruktur gebunden. Dadurch ist eine konkrete Coroutinen-Implementation wie in Quelltext 4.4 möglich.

Tatham hat in seiner weiteren Arbeit auf dieser Grundidee aufbauende, generische Coroutinen realisiert, deren konkrete Implementation durch Präprozessormakros verborgen wird. Hervorzuheben an dieser Lösung ist die Tatsache, dass sie, im Rahmen der Programmiersprache C, portabel ist.

Ablaufsteuerung durch Registersatzsicherung

Ein Ansatz in hardwarenahen[2] Sprachen besteht darin, die momentanen Inhalte aller relevanten Register des Prozessors (einen sogenannten *Registersatz*) zu sichern und bei einer Fortsetzung der Coroutine wiederherzustellen.

Unter den gesicherten Registerinhalten spielt der des Programmzählers eine besondere Rolle. Prinzipiell erfolgt bei diesem die Sicherung und Wiederherstellung analog zu den übrigen Registern. Wenn jedoch bei der Wiederherstellung der Wert des Programmzählers durch einen vorher gesicherten Wert überschrieben wird, löst dies automatisch einen Sprung hinter die letzte, vor der Unterbrechung bearbeitete Anweisung der Coroutine aus, was exakt dem gewünschten Verhalten bei der Coroutinenfortsetzung entspricht.

Je nachdem, wie die Registersatzsicherung konkret umgesetzt worden ist, lassen sich durch diesen Ansatz sowohl symmetrische als auch asymmetrische Coroutinen realisieren. Bei symmetrischen Coroutinen wird bei jedem Steuerungswechsel der entsprechende Registersatz direkt durch den der nachfolgenden Coroutine ersetzt. Bei einer Implementation für asymmetrische Coroutinen wird hingegen bei jeder Unterbrechung lediglich der Registersatz gesichert. Anschließend verhält sich das Programm jedoch so, als wäre es aus einem Subroutinenaufruf zurückgekehrt; erst bei einer Fortsetzung der Coroutine wird ein gespeicherter Registersatz wiederhergestellt.

Ein Beispiel für die Ablaufsteuerung durch Registersatzsicherung stellt die Implementation von Fischer und Ahrens dar. Diese Coroutinen-Implementation, die die Verwendung von symmetrischen und stackerhaltenden Coroutinen ermöglicht, bildet die Grundlage der Simulationsbibliotheken ODEM [FA96] und ODEMx [Ger03]. Während der Großteil beider Simulationsbibliotheken in C++ implementiert wurde, erfolgte die gemeinsame Coroutinen-Implementation wegen der benötigten Hardwarenähe in C. Die Registersicherung basiert in dieser Implementation auf zwei Standard-C-Funktionen: *setjmp* und *longjmp*. Dieses Paar von Funktionen, das normalerweise zur Implementation von Ausnahmebehandlungen in der Programmiersprache C benutzt wird, erfüllt exakt die benötigten Aufgaben der Registersatzsicherung und -wiederherstellung.

[2] Als *hardwarenah* sollen in dieser Arbeit alle Programmiersprachen gelten, mittels derer sich zur Laufzeit der damit erstellten Programme ein direkter, lesender und schreibender Zugriff sowohl auf den Hauptspeicher als auch auf einzelne Register des Prozessors realisieren lässt.

Implementation der Kontextsicherung

Kontextsicherung durch explizites Sichern lokaler Variablenwerte

Ein einfacher Ansatz der Kontextsicherung besteht darin, zu jeder lokalen Variable eine globale zu definieren und bei jeder Unterbrechung der Coroutine den Wert der lokalen Variable in die zugehörige globale zu kopieren. Bei jeder Fortsetzung wiederum werden zuerst die Werte der lokalen Variablen aus den globalen Kopien wiederhergestellt.

Diese Umsetzung ließe sich prinzipiell von Hand implementieren; praktikabel einsetzbar wird sie jedoch erst, wenn die entsprechenden Quelltextpassagen für das Sichern und Wiederherstellen der Variablenwerte automatisch generiert werden. Allerdings ist mit letztgenanntem Ansatz die aufwendige Implementation eines Parsers für die Simulationssprache verbunden, der die zu sichernden Variablen ermittelt. Bei Simulationsbibliotheken würde dies die Implementation eines Parsers für eine vollständige Programmiersprache bedeuten.

Die durch diese Art der Kontextsicherung umgesetzte Continuation ist offensichtlich lokal und einfachfortsetzbar. Eine Erweiterung auf mehrfachfortsetzbare Continuations ist jedoch vergleichsweise einfach möglich, wenn bei jeder Unterbrechung die Sicherung in einem jeweils neuen Satz globaler Variablen erfolgt, also die Wiederherstellung von Werten aus älteren Variablensätzen weiterhin möglich ist. Analog lassen sich auch die im Folgenden vorgestellten Kontextsicherungsverfahren mehrfachfortsetzbar gestalten.

Kontextsicherung durch Vermeidung lokaler Variablen

Eine alternative Umsetzung der Kontextsicherung besteht im völligen Verbot lokaler Variablen. Als Ersatz erhält jede Coroutine einen zusätzlichen Parameter in Form einer Referenz auf einen Speicherbereich, in dem die Coroutine „lokale“ Daten speichern darf.

Der größte Nachteil der Kontextsicherung durch Vermeidung lokaler Variablen besteht in den strikten Regeln, an die sich ein Nutzer einer solchen Coroutinen-Implementation halten muss: Zugriffe auf lokale Variablen sind nur über die gegebene Referenz möglich und bei jedem Aufruf muss die jeweils korrekte Referenz mitgegeben werden. Dennoch wird dieser Ansatz aufgrund seiner vergleichsweise einfachen Implementation in verschiedenen Coroutinen-Implementationen, z. B. der bereits erwähnten von Tatham [Tat00], verwendet.

Eine Variante der Vermeidung lokaler Variablen ist in Programmiersprachen möglich, die das Konzept globaler Variablen mit lokalem Gültigkeitsbereich anbieten. Ein Beispiel dafür sind die als *static* deklarierten Variablen in C und C++. Die Verwendung derartiger Variablen ermöglicht zwar auf sehr einfache Art die Kontextsicherung zwischen zwei Coroutinenaufrufen, allerdings ist die dabei implizit genutzte Continuation offensichtlich nur einfachfortsetzbar.

Kontextsicherung durch Stackmanipulation

In stackbasierten Sprachen besteht ein weiterer Ansatz zur Lösung der Kontextsicherung in der Implementation von Stackmanipulationen. Dieser Ansatz basiert auf der Tatsache, dass sich zum Zeitpunkt der Unterbrechung einer Coroutine die Werte aller zu sichernden lokalen Variablen im Stacksegment des aktuellen Coroutinenaufrufes befinden. Eine Sicherung dieses Stacksegmentes unmittelbar vor der Unterbrechung einer Coroutine und seine Wiederherstellung zum Zeitpunkt der Fortsetzung würde also automatisch die Sicherung und Wiederherstellung der Belegungen aller lokalen Variablen bedeuten.

Dabei muss eine der beiden folgenden Bedingungen erfüllt sein, um die Konsistenz des Programmes nicht zu gefährden:

- Wenn, wie beschrieben, nur das Stacksegment des aktuellen Coroutinenaufrufes gesichert/wiederhergestellt wird, sind alle Aufrufe zur Unterbrechung derselbigen nur innerhalb der Coroutine selbst erlaubt, also nicht innerhalb tiefer verschachtelter Subroutinen. Diese Coroutinen-Implementation ist also nicht stackerhaltend im Sinne von Abschnitt 4.3. Sollte dennoch ein tiefer verschachtelter Unterbrechungsaufruf erfolgen, führt dies bei dieser Implementation zu einer fehlenden Sicherung aller tiefer verschachtelten lokalen Variablen und spätestens bei einem späteren Zugriff zu einer nur schwer behebbaren Inkonsistenz im Programm.
- Modifiziert man hingegen den beschrieben Ablauf dahingehend, dass nicht nur das Segment des aktuellen Coroutinenaufrufes, sondern auch alle diesem untergeordneten Segmente gesichert und später wiederhergestellt werden, erhält man eine stackerhaltende Coroutinen-Implementation.

Analog zur Sicherung/Wiederherstellung der Registersätze hängt auch von der konkreten Implementation der Sicherung/Wiederherstellung der Stacksegmente ab, ob dadurch symmetrische oder asymmetrische Coroutinen realisiert werden. Bei symmetrischen Coroutinen werden Stacksegmente stets durch ein unmittelbar aufeinanderfolgendes Sichern und Wiederherstellen substituiert. Bei asymmetrischen Coroutinen hingegen erfolgen die Stacksegmentsicherung und -wiederherstellung getrennt voneinander in den beiden Methoden für das Unterbrechen und Fortsetzen von Coroutinen.

Ein Beispiel für die Kontextsicherung durch Stackmanipulation findet sich in der oben bereits erwähnten Implementation von Fischer und Ahrens [FA96]. Die Stacksicherung erfolgt hier durch ein direktes Kopieren des Speicherbereichs, an dem sich die entsprechenden Stacksegmente befinden, mittels der Standard-C-Funktion *memcpy*.[3]

[3] Damit diese Kopieroperation durchgeführt werden kann und gleichzeitig portabel ist, müssen vorher zwei Informationen bekannt sein: der Anfang des entsprechenden Coroutinenstacksegmentes und die Richtung, in der der Stack auf der konkreten Hardware wächst. Beides wird dadurch ermittelt, dass jede neu gestartete Coroutine sofort zwei lokale Variablen anlegt: Die Speicheradresse der ersten Variable entspricht zwangsläufig dem Anfang des Stacksegments der Coroutine; aus der

Implementation durch nebenläufige Programmierung

Es existiert ein weiterer prinzipieller Ansatz zur Implementation von Coroutinen, der sich nicht in die strikte Aufgabenverteilung der beiden vorherigen Abschnitte einsortieren lässt, da er gleichzeitig sowohl das Problem der Ablaufsteuerung als auch der Kontextsicherung löst. Dabei handelt es sich um die zwangsserialisierte Ausführung eigentlich nebenläufiger Mechanismen der zugrundeliegenden Programmiersprache (z. B. Threads in Java) bzw. des zugrundeliegenden Betriebssystems (z. B. Windows- oder UNIX-Prozesse). Im Folgenden wird der Einfachheit halber ausschließlich der Terminus *Thread* benutzt, die Ausführungen lassen sich aber auch mit anderen nebenläufigen Strukturen umsetzen.

Die Grundidee dieses Ansatzes besteht darin, zunächst jede Coroutine in einem separaten Thread zu implementieren. Auch das Hauptprogramm wird in Form eines Threads programmiert, so dass es später, während der Ausführung jederzeit unterbrech- und fortführbar ist.

Soll nun während der Laufzeit des Programmes eine Coroutine erstmalig gestartet werden, so wird der entsprechende Coroutinenthread durch das Hauptprogramm gestartet. Unmittelbar nach dieser Coroutinen/Thread-Aktivierung unterbricht sich das Hauptprogramm selbst, so dass ab sofort nur noch die Anweisungen der Coroutine abgearbeitet werden. Sollte sich diese Coroutine nun ihrer Unterbrechung nähern, so aktiviert sie den Coroutinenthread der nachfolgenden Coroutine (bei symmetrischen Coroutinen) bzw. des Hauptprogrammes (bei asymmetrischen Coroutinen) und unterbricht sich unmittelbar danach selbst, so dass anschließend wieder genau ein Thread läuft. Im Endeffekt erhält man also ein prinzipiell nebenläufig implementiertes Programm, deren einzelne nebenläufige Strukturen jedoch nie parallel ablaufen (von den wenigen Anweisungen während eines Coroutinenwechsels abgesehen).

Dieser Implementationsansatz besitzt mehrere Vorteile: Zunächst löst er gleichzeitig das Problem der Ablaufsteuerung und der Kontextsicherung, da die Thread-Implementation diese Aufgaben übernimmt, und ist dadurch auch mit vergleichsweise geringem Aufwand umsetzbar. Wie bereits angedeutet, lassen sich mit diesem Ansatz sowohl symmetrische als auch asymmetrische Coroutinen realisieren. Darüber hinaus sind die implementierten Coroutinen unbeschränkt und stackerhaltend.

Ein Nachteil dieses Ansatzes besteht in der Abhängigkeit von der benutzten Implementation der Nebenläufigkeit. Existieren in einer Programmiersprache keine nebenläufigen Konzepte und lassen sich diese auch nicht ohne weiteres (z. B. durch Unterstützung des Betriebssystems) nachrüsten, so ist der gesamte Ansatz hinfällig. Auch leidet die Performanz einer durch Threads umgesetzten Coroutinen-Implementation, wenn die zugrundeliegende Thread-Implementation selbst nicht performant ist.

Differenz zwischen dieser Adresse und der Adresse der zweiten lokalen Variable lässt sich die Wuchsrichtung des Stacks ermitteln.

Implementation von Coroutinen in Java

Für das Ziel dieser Arbeit, die Implementation einer optimistisch-parallelen prozessorientierten Simulationsbibliothek in Java, wird eine Coroutinen-Implementation in eben dieser Sprache benötigt. Allerdings lässt sich nur ein Teil der in den vorangegangenen Abschnitten vorgestellten Ansätze zur Coroutinen-Implementation direkt in dieser Programmiersprache umsetzen.

Bezüglich der Ablaufsteuerung lässt sich festhalten, dass sich mangels entsprechender Sprachmittel weder die angesprochene Umsetzung in Assembler noch die Implementation durch unbedingte Sprünge direkt in Java realisieren lassen [AGH06]. Auch das Ersetzen von Sprunganweisungen durch Mehrfachverzweigungen analog zur Umsetzung von Tatham lässt sich in Java nicht durchführen, da die Fallanweisungen innerhalb einer Mehrfachverzweigungen in Java an die Blockstruktur gebunden sind (d. h., jedes *case* darf nur im direkten Kontext der zugehörigen *switch*-Anweisung auftreten). Die Variante der Registersatzsicherung scheidet aufgrund des Aufbaus der virtuellen Maschine von Java [LY99], die keine der zwingend benötigten Zugriffsmöglichkeiten auf die einzelnen Register vorsieht, als Implementationsansatz ebenfalls aus.

Günstiger sieht die Situation bezüglich der Kontextsicherung aus: Sowohl die angesprochenen Varianten der expliziten Sicherung lokaler Variablenwerte als auch die der Vermeidung lokaler Variablen lassen sich direkt in Java umsetzen. Der von den damit verbundenen Vor- und Nachteilen her günstigste Ansatz der Kontextsicherung durch Stackmanipulation scheitert hingegen erneut durch mangelnde Zugriffsmöglichkeiten auf die virtuelle Maschine während der Laufzeit eines Java-Programmes.

Implementation durch zwangsserialisierte Threads

Den vielversprechendsten Ansatz in Java scheint die Coroutinen-Implementation durch Ausnutzung von Nebenläufigkeiten darzustellen, da die Programmiersprache bereits in ihrer ursprünglichen Sprachdefinition eine Unterstützung paralleler Ablaufstrukturen in Form von Threads besitzt. Des Weiteren löst dieser Ansatz, wie in Abschnitt 4.6 beschrieben, gleichzeitig das Problem der Ablaufsteuerung und der Kontextsicherung.

Wie ebenfalls bereits beschrieben, lassen sich durch das Verfahren zwangsserialisierter Threads sowohl symmetrische als auch asymmetrische Coroutinen realisieren. Da sich bereits aus den Ausführungen in Abschnitt 4.3 ergeben hat, dass asymmetrische Coroutinen eher für das Ziel dieser Arbeit geeignet sind, wird im Folgenden der Schwerpunkt auf eben dieser Art von Coroutinen liegen.[4]

In Quelltext 4.5 ist die Realisierung einer asymmetrischen Coroutinenklasse zu sehen. Wie bei einer asymmetrischen Coroutinen-Implementation zu erwarten, ist das Unterbrechen und Fortsetzen der Coroutine in zwei separaten Methoden (*suspend* und *resume*) realisiert worden. Dabei spiegelt sich die Asymmetrie auch in der

[4] Eine sehr detaillierte Darstellung der Implementation symmetrischer Coroutinen in Java inklusive einer Diskussion verschiedener Implementationsvarianten findet sich in [Hel00].

```
1  public class MySemiCoroutine implements Runnable {
2
3      private Thread thread;
4
5      public void run() {
6          [...] // der eigentliche Coroutinenquelltext
7      }
8
9      /* Starten/Fortsetzen einer Coroutine */
10     public synchronized void resume() {
11         if (thread == null) {
12             // erstmaliges Starten der Coroutine
13             thread = new Thread(this); // erstelle neuen Coroutinen-Thread
14             thread.setDaemon(true);  // Thread soll mit Hauptprogramm enden
15             thread.start();          // starte Coroutinen-Thread
16         }
17         else {
18             // Fortsetzen der Coroutine
19             synchronized (thread) {
20                 thread.notify();     // setze Coroutinen-Thread fort
21             }
22         }
23         try {
24             this.wait();             // unterbreche Hauptprogramm-Thread
25         } catch (InterruptedException e) {}
26     }
27
28     /* Unterbrechen einer Coroutine */
29     private void suspend() {
30         synchronized (thread) {
31             synchronized (this) {
32                 this.notify();       // setze Hauptprogramm fort
33             }
34             try {
35                 thread.wait();       // unterbreche Coroutinen-Thread
36             } catch (InterruptedException e) {}
37         }
38     }
39
40 }
```

Quelltext 4.5: Asymmetrische Coroutinen-Implementation in Java mittels zwangsserialisierter Threads

Sichtbarkeit der Methoden wider: Da eine Unterbrechung nur von der Coroutine selbst ausgelöst werden darf, ist die entsprechende Methode privat, während die von außen zu rufende Fortsetzungsmethode öffentlich deklariert worden ist. Die Implementation der eigentlichen Coroutinenaufgabe erfolgt in der Methode *run*.

Das erstmalige Starten der Coroutine erfolgt bei der gezeigten Implementation ebenfalls durch einen Aufruf von *resume*. Diese Methode erzeugt dann als erstes ein neues Thread-Objekt, wobei eine Referenz auf dieses Objekt in der Variable *thread* gespeichert wird. Anschließend wird an diesem Thread-Objekt die *Daemon*-Eigenschaft gesetzt, was dafür sorgt, dass dieser Thread automatisch terminiert wird, wenn das Hauptprogramm endet. Anschließend wird der Thread gestartet, d. h., in diesem neuen Thread wird nun der Inhalt von *run* ausgeführt, während der Hauptprogramm-Thread mittels *wait* unterbrochen wird.[5]

Ab diesem Zeitpunkt läuft nur noch der Coroutinenthread, bis sich dieser mittels Aufrufes von *suspend* unterbricht. Wie im Quelltext zu sehen, führt dies zur Reaktivierung des Hauptprogrammes mittels *notify* und unmittelbar danach zur Unterbrechung des Coroutinenthreads durch den Aufruf von *wait*. Bei einer erneuten Fortsetzung der Coroutine seitens des Hauptprogrammes durch den Aufruf von *resume* erfolgt lediglich die Fortsetzung desjenigen Threads, von dem eine Referenz beim erstmaligen Aufruf in der Variable *thread* hinterlegt wurde. Anschließend wird wieder wie beim erstmaligen Aufruf von *resume* der Thread des Hauptprogrammes unterbrochen.

Während die prinzipielle Arbeitsweise der Coroutinen-Implementation im Quelltext vergleichsweise einfach nachvollziehbar ist, sind die spezifischen Synchronisationsmechanismen der Java-Threads etwas komplizierter. Die Grundlage für die Synchronisation von Java-Threads bildet das sogenannte Monitorkonzept. Ein Monitor stellt dabei einen, nicht notwendigerweise zusammenhängenden, Quelltextteil dar, dessen Zugang durch einen binären Semaphor geschützt ist. Das heißt, jeder Thread, der den Monitor ausführen soll, muss zunächst den zugehörigen Semaphor setzen (im Java-Sprachgebrauch: Besitzer des Monitors werden) und darf erst dann mit der Abarbeitung des Monitors beginnen. Ist der Semaphor bereits gesetzt, d. h., der Monitor wird gerade von einem anderen Thread ausgeführt, so muss ein weiterer Thread warten, bis der gerade aktive den Monitor freigibt.

Zur Markierung von Monitoren wird in Java die Anweisung *synchronized* benutzt, die als Parameter die Referenz auf ein Objekt erwartet, das als Semaphor benutzt wird, d. h., alle mit dem gleichen Objekt markierten Monitore werden bei Betreten eines derartigen Monitors ebenfalls gesperrt – ansonsten hat die Benutzung als Semaphor keine weiteren Auswirkungen auf das Objekt. Eine häufig benutzte Abkürzung stellt die Verwendung eines parameterlosen *synchronized* als Attribut für eine Methode dar, wie im Quelltext bei der Methode *resume* zu sehen ist. In die-

[5] Eine der größeren Hürden beim Verständnis von Java-Threads besteht darin, dass die Methoden innerhalb eines Thread-Objektes mitnichten nur in/von diesem Thread bearbeitet werden. Im konkreten Beispiel wird *resume* immer im Thread des Hauptprogrammes ausgeführt.

sem Fall verhält sich die Methode, als wäre ihr Körper von einem *synchronized(this)* umschlossen.

Bei der kontrollierten Unterbrechung und Fortsetzung von Threads werden in Java ebenfalls normale Objekte zur Synchronisation benutzt. Zu diesem Zweck existieren die Methoden *wait* und *notify*, die in *java.lang.Object* endgültig (*final*) definiert wurden und damit in jedem Java-Objekt unverändert zur Verfügung stehen. Sobald an einem Java-Objekt die Methode *wait* von einem Thread gerufen wird, stellt dieser umgehend seine Arbeit ein und läuft erst dann wieder weiter, wenn, zwangsläufig durch einen anderen Thread, am selben Objekt die Methode *notify* gerufen wird. Dabei gilt die Bedingung, dass welcher Thread auch immer an einem Objekt *wait* oder *notify* rufen möchte, zu diesem Zeitpunkt den Monitor dieses Objektes besitzen muss. Zusätzlich gilt die spezielle Regel, dass ein mittels *wait* suspendierter Thread den zugehörigen Monitor für die Zeit der Unterbrechung freigibt, ansonsten wäre die Reaktivierung durch einen anderen Thread unmöglich.

In Quelltext 4.5 werden zwei Threads über zwei Monitore basierend auf zwei Objekten (der Instanz von *MySemiCoroutine*, referenziert durch *this*, und der Instanz des Coroutinenthreads, referenziert durch die Variable *thread*) miteinander synchronisiert.

Bewertung der Prozessimplementation durch zwangsserialisierte Threads

Die gezeigte, Thread-basierte Implementation ermöglicht die Verwendung asymmetrischer, unbeschränkter und stackerhaltender Coroutinen in Java. Wie bereits beschrieben, löst sie nicht nur automatisch die Coroutinenaufgaben der Ablaufsteuerung und Kontextsicherung, sondern ist darüber hinaus auch noch vergleichsweise einfach zu implementieren. Die genannten Eigenschaften sind auch der Grund, warum die Implementation von Coroutinen durch zwangsserialisierte Threads das Standardverfahren bei der Erstellung sequentieller, prozessorientierter Simulationsbibliotheken in Java darstellt, wie z. B. in DESMO-J [LP99], JAVASIMULATION [Hel00] und JDISCO [Hel01], in den beiden letztgenannten Bibliotheken in Form symmetrischer Threads.

Allerdings ist dieses Verfahren nicht zur Implementation von Simulationsprozessen in optimistisch-parallelen Simulationen geeignet. Der Grund dafür liegt letztlich in der Thread-Implementation von Java: Sobald ein Java-Thread fortgesetzt wird, geht sein vorher gespeicherter Zustand verloren, so dass es unmöglich ist, den Thread von diesem Zustand aus erneut fortzusetzen. Es ist auch nicht möglich, den gespeicherten Zustand vorher zu kopieren, da Java keine Möglichkeit bietet, diesen überhaupt zu referenzieren.

Diese Einfachfortsetzbarkeit von Java-Threads sorgt dafür, dass jede auf der Zwangsserialisierung von Java-Threads basierende Coroutinen/Continuation-Implementation ebenfalls nur einfachfortsetzbare Continuations umsetzen kann. Da jedoch, wie in Abschnitt 4.5 beschrieben, mehrfachfortsetzbare Continuations zur Realisierung von optimistisch-parallelen Simulationsprozessen zwingend erforder-

lich sind, scheidet die Implementation durch zwangsserialisierte Java-Threads als Implementationsansatz aus.

4.7 Continuation-Implementation durch Modifikation des Bytecodes

Nachdem in den vergangenen Abschnitten verschiedene Coroutinen-Implementationen vorgestellt und jeweils anschließend die dabei implementierten Continuations untersucht wurden, werden in diesem Abschnitt originäre Continuation-Implementationen analysiert.

Wie die Ausführungen des vorherigen Abschnitts ergeben haben, ist die Implementation einer Coroutine/Continuation in Java, von der Thread-Implementation abgesehen, nur auf Umwegen möglich. Dabei stellt insbesondere die Implementation der Ablaufsteuerung das Kernproblem dar, da die für eine elegante Umsetzung benötigten Sprachmittel und/oder Speicherzugriffsmöglichkeiten zur Laufzeit in Java bzw. in der Java Virtual Machine nicht gegeben sind.

Dennoch existieren zwei Continuation-Implementationen für Java, die, obwohl sie den gleichen Einschränkungen unterworfen sind, verschiedene der bereits vorgestellten Implementationsansätze in eben dieser Sprache umsetzen: RIFE/CONTINUATIONS und JAVAFLOW. Im Folgenden werden zunächst der gemeinsame Hintergrund der beiden Continuation-Implementationen sowie die grundsätzlichen Eigenschaften der von diesen umgesetzten Coroutinen/Continuations vorgestellt, bevor der gemeinsame Implementationsansatz, die nachträgliche Modifikation des Bytecodes, im Detail diskutiert wird.

Continuations im Anwendungsgebiet von Webapplikationen

Coroutinen/Continuations spielen seit mehreren Jahren eine wichtige Rolle bei der Implementation von Webapplikationen [Tat06, Que03, Que04]. Der Grund dafür ist gut nachvollziehbar, wenn man die Arbeitsweise einer Webapplikation und die daraus folgenden Implementationsansätze näher betrachtet.

Zunächst unterscheidet sich eine Webapplikation von einem normalen Programm in zwei zentralen Punkten. Zum einen wird sie zur Laufzeit auf einem Webserver ausgeführt und zum anderen erfolgt bei dieser Ausführung jegliche Interaktion mit dem Benutzer indirekt durch einen Webbrowser auf seinem Rechner.

Eine sehr einfache Herangehensweise an die Entwicklung von Webapplikationen besteht darin, den späteren Benutzer als eine Art responsives System zu betrachten. Die eigentliche Implementation hat dann eine sehr große Ähnlichkeit mit der Implementation des Lebenszyklus' eines Prozesses (vergleiche Abschnitt 2.8).

Prinzipiell erfüllt eine Webapplikation drei Teilaufgaben, die sich in jeder Iteration einer Endlosschleife wiederholen:

- das Schicken von Daten an den Web-Browser des Nutzers,

- das Warten auf eine Reaktion des Nutzers und
- das Verarbeiten der vom Nutzer erhaltenen Daten.

Die Ähnlichkeit zwischen den Implementationen einer Webapplikation und eines Simulationsprozesses ist relativ offensichtlich. Beide basieren auf einer Endlosschleife, deren Abarbeitung durch eine Warteanweisung unterbrochen wird. Ein zentraler Unterschied besteht jedoch darin, dass eine Verwendung von Coroutinen bei der Implementation von Webapplikationen zwar möglich, aber nicht zwingend nötig ist, da die Reaktion des Nutzers nicht vom gleichen Programm „ausgeführt“ werden muss.

Es gibt jedoch eine Verhaltensmöglichkeit des Nutzers, die mit der beschriebenen prinzipiellen Implementation von Webapplikationen inkompatibel ist: die Nutzung des sogenannten „Back-Buttons“. Als diese Schaltfläche in Webbrowsern eingeführt wurde, gab es noch keine dynamisch generierten Webseiten oder gar Webapplikationen, sondern lediglich statische, untereinander verlinkte Webseiten, die durch das Folgen der Verlinkung nacheinander angezeigt werden konnten.

Der Back-Button ermöglicht es, den letzten Schritt entlang einer Verlinkung rückgängig zu machen, d. h., der Nutzer sieht nach dem Betätigen dieser Schaltfläche wieder die letzte Seite, die er vor der aktuellen angezeigt bekam. Umgesetzt wird der Back-Button, indem sich der Browser jede besuchte URL in einer Liste speichert und bei Bedarf einfach die vorletzte URL der Liste erneut aufruft.

Für eine Webapplikation, die mit dem Nutzer auf der Basis dynamisch generierter Webseiten interagiert, stellt die Existenz des Back-Buttons eine zusätzliche Herausforderung dar.[6] Der Nutzer erwartet, geprägt von seinen Erfahrungen mit regulären Webseiten, dass bei einer Betätigung dieser Schaltfläche seine letzte Aktion rückgängig gemacht wird. Für den Webapplikations-Entwickler bedeutet dies, dass die Webapplikation in diesem Fall in den Zustand überführt werden muss, den sie vor der letzten Interaktion mit dem Nutzer hatte. Unter Verwendung der Simulationsterminologie könnte man auch formulieren, dass die Webapplikation einen Time-Warp zum letzten gültigen Zustand durchführen muss.

An dieser Stelle ermöglichen Continuations einen eleganten Implementationsansatz. Man muss die bisher beschriebene Implementation lediglich an drei Stellen modifizieren:

- die Webapplikation erhält eine Datenstruktur in Form einer Liste von Continuations,
- unmittelbar vor jeder Ausgabe an den Nutzer wird der aktuelle Laufzeitzustand in Form einer Continuation als neues Element der Liste gespeichert und
- unmittelbar nach jeder Reaktion des Nutzers prüft die Webapplikation, ob es sich bei der Reaktion um die Betätigung des Back-Buttons handelte. Im Nega-

[6] Es soll nicht unerwähnt bleiben, dass es etliche Webapplikationen (auch auf den Webauftritten namhafter Firmen) gibt, deren Programmierer sich dieser Herausforderung vollständig verweigern. Zwei (leider) übliche Ansätze bestehen darin, den Back-Button wahlweise programmatisch zu deaktivieren oder aber dem Nutzer direkt im angezeigten Text mitzuteilen, dass er diese Schaltfläche nicht benutzen möge.

```
1  public class MyCoroutine implements Runnable {
2      @Override
3      public void run() {
4          int number = 0;
5          while (true) {
6              number++;
7              System.out.println(number);
8              Continuation.suspend();
9          }
10     }
11 }
```

Quelltext 4.6: Java-Beispiel für die Implementation einer Coroutine in JAVAFLOW

tivfall fährt sie fort wie bisher; im Positivfall setzt sie erneut mit der vorletzten Continuation der Liste fort.

Dieser Ansatz ist relativ einfach zu implementieren und genau in dieser Tatsache liegt auch ein wesentlicher Grund dafür, dass die beiden im folgenden Abschnitt beschriebenen Continuation-Implementationen ihren Ursprung im Bereich der Webapplikations-Entwicklung haben.

Existierende Continuation-Implementationen in Java

Es existieren zwei bekanntere Continuation-Implementationen für Java: RIFE/CONTINUATIONS und JAVAFLOW. Wie bereits erwähnt, stammen beide Continuation-Implementationen aus dem Bereich der Webapplikationsentwicklung und basieren ebenfalls beide auf dem gleichen Implementationsansatz. Bevor dieser jedoch im Detail beschrieben wird, sollen zunächst die beiden Continuation-Implementationen kurz einzeln vorgestellt werden.

RIFE/CONTINUATIONS

Bei RIFE/CONTINUATIONS [RIF] handelt es sich um ein Projekt, das ursprünglich als eigenständiges Teilprojekt des Webapplikationsframeworks RIFE entstanden ist. Inzwischen wurde RIFE/CONTINUATIONS vollkommen in den Entwicklungsprozess von RIFE integriert, so dass der Name RIFE/CONTINUATIONS obsolet ist. Im Rahmen dieser Arbeit wird er aber zur Abgrenzung vom Gesamtprojekt RIFE weiterverwendet.

Zur Implementation des Bytecode-Rewritings verwendet RIFE/CONTINUATIONS das Bytecode-Manipulations und -Analyse-Framework ASM [BLC02, OW2], das auch in einer Vielzahl anderer Java-Projekte (z. B. ASPECTJ, JYTHON, ...) benutzt wird.

Auch jenseits der eigenen Implementation sind die Entwickler von RIFE/CONTINUATIONS an der Propagierung von Continuations in Java beteiligt. So sind in dem

Entwicklerkreis des RIFE/CONTINUATIONS-Projektes verschiedene Veröffentlichungen zu diesem Thema entstanden [Bev07], unter denen sich auch ein Java Specification Request (JSR) befindet, der Continuations als nativen Teil der Sprache Java in einer zukünftigen Version vorschlägt [Bev08].

Die bei Verwendung von RIFE/CONTINUATIONS implementierten Coroutinen sind asymmetrisch und unbeschränkt. Die dabei eingesetzten Continuations sind lokal und mehrfachfortsetzbar. Darüber hinaus bietet RIFE/CONTINUATIONS sogar eine automatische Sicherung/Wiederherstellung nicht nur der Coroutinen-lokalen Variablenwerte, sondern sogar der Attribute der Objekte, in denen die Coroutinen implementiert werden. In einer Simulationsumsetzung würde dies bedeuten, dass automatisch alle Zustandsvariablen eines Prozesses gesichert und wiederhergestellt werden.

Alle bis hier beschriebenen Eigenschaften lassen auf eine gute Eignung von RIFE/CONTINUATIONS für die Implementation optimistisch-paralleler Prozesse schließen. Es muss jedoch festgehalten werden, dass RIFE/CONTINUATIONS nur die Implementation nicht-stackerhaltender Coroutinen ermöglicht. Wie in Abschnitt 4.3 beschrieben, bedeutet dies, dass derartige Coroutinen keine *suspend*-Aufrufe aus Unterroutinen erlauben. Laut der Dokumentation von RIFE/CONTINUATIONS ist diese Einschränkung (die nicht nur Coroutinen im Simulationsumfeld betrifft) bekannt und soll in einer zukünftigen Version beseitigt werden. Bis dahin scheidet jedoch RIFE/CONTINUATIONS als Implementationsgrundlage für optimistisch-parallele Prozesse aus.

JAVAFLOW

JAVAFLOW [ORCAE06] ist entstanden als ein Teilprojekt des APACHE-COMMONS-Projektes [War06, Apaa], das die Entwicklung vielfältig einsetzbarer Klassenbibliotheken zum Ziel hat. Wie auch RIFE/CONTINUATIONS hat JAVAFLOW die konkrete Implementation der Bytecode-Modifikation mit Hilfe einer bereits vorher existierenden Bytecode-Manipulationsbibliothek umgesetzt und zwar der BYTE CODE ENGINEERING LIBRARY (BCEL) [Apab].

Im Gegensatz zu RIFE/CONTINUATIONS besitzen mit JAVAFLOW implementierte Coroutinen/Continuations alle Eigenschaften, die zur Implementation von optimistisch-parallelen Prozessen benötigt werden (siehe Abschnitt 4.5). Die mit JAVAFLOW umgesetzten Coroutinen sind asymmetrisch, stackerhaltend und unbeschränkt; die Continuations sind lokal und mehrfachfortsetzbar.

Daher bildete JAVAFLOW auch die Grundlage der Prozessumsetzung bei der Implementation des im Rahmen dieser Arbeit entstandenen Simulationskerns. In den folgenden Abschnitten wird zunächst die Verwendung von JAVAFLOW aus Sicht des Endnutzers vorgestellt, bevor die konkrete Implementationsgrundlage der Bytecode-Modifikation im Detail beschrieben wird.

```
1  public class MyContinuationStarter {
2      public static void main(String[] args) throws Exception {
3
4          MyCoroutine mc = new MyCoroutine();
5
6          /* Erstes Starten */
7          Continuation cont1 = Continuation.startWith(mc);
8
9          /* Erstes Fortsetzen */
10         Continuation cont2 = Continuation.continueWith(cont1);
11
12         /* Zweites Fortsetzen (ohne Speicherung der neuen Continuation) */
13         Continuation.continueWith(cont2);
14
15         /* Erneutes Fortsetzen der ersten Continuation */
16         /* (Demonstration der Mehrfachfortsetzbarkeit) */
17         Continuation.continueWith(cont1);
18     }
19 }
```

Quelltext 4.7: Starter-Klasse für die Coroutine aus Quelltext 4.6

Coroutinen/Continuations in JAVAFLOW aus Nutzersicht

In JAVAFLOW erfolgt die Umsetzung einer Coroutine in einer Klasse, die das Interface *Runnable* implementiert. Diese eigentlich aus dem Java-Thread-Kontext stammende Schnittstelle verlangt die Implementation einer Methode *run*, die im Fall von JAVAFLOW die Coroutine beherbergen soll. Nur aus derartigen *run*-Methoden (und den von diesen aus aufgerufenen Methoden) heraus sind Aufrufe an die Klassenmethode *Continuation.suspend* möglich, die für die fortsetzbare Unterbrechung der jeweils rufenden Methode sorgt.

In Quelltext 4.6 ist die Implementation einer einfachen Coroutine zu sehen. Dabei handelt es sich um die JAVAFLOW-Umsetzung der Pseudocode-Coroutine aus Quelltext 4.2, die bei jeder Fortsetzung eine um eins inkrementierte Zahl ausgibt und sich jeweils unmittelbar danach selbst unterbricht.

Gestartet wird eine JAVAFLOW-Coroutine durch den Aufruf der statischen Methode *Continuation.startWith*, die als Parameter eine Referenz auf eine Instanz einer wie oben beschriebenen Klasse enthält. Sobald sich die Coroutine selbst durch *Continuation.suspend* unterbricht, kehrt der Aufruf von *Continuation.startWith* zurück und liefert als Rückgabewert eine Instanz der Klasse Continuation, die den Bearbeitungsstand der Coroutine zum Zeitpunkt der Unterbrechung repräsentiert.

Mit Hilfe der statischen Methode *Continuation.continue* können unterbrochene Coroutinen/Continuations fortgesetzt werden, indem man eine Referenz auf die jeweils fortzusetzende Continuation als Parameter übergibt. Nach jeder Fortsetzung erhält man eine neue Continuation, während die als Parameter übergebene Continuation unverändert bleibt und bei Bedarf ebenfalls (erneut) fortgesetzt werden kann (die Continuation-Implementation von JAVAFLOW ist mehrfachfortsetzbar).

In Quelltext 4.7 ist ein einfaches Java-Programm zu sehen, das zuerst die in Quelltext 4.6 gezeigte Coroutine startet und dann mehrmals fortsetzt. Dieses Programm beinhaltet auch eine Demonstration der Mehrfachfortsetzbarkeit: ein einmal gespeicherter Laufzeitzustand (*cont1*) wird zweimal fortgesetzt (Zeile 10+17).

Aus Endnutzersicht erfolgt die Coroutinen/Continuation-Implementation unter der ausschließlichen Verwendung von Sprachmitteln, die durch die Spezifikation der Programmiersprache Java zur Verfügung gestellt werden. Die konkrete Implementation, insbesondere der Unterbrechungs- und Fortsetzungsmethoden bleibt ihm, zumindest im Quelltext, verborgen.

Dennoch erfolgt die Implementation für den Endnutzer einer JAVAFLOW-Coroutine nicht vollkommen transparent, da der Übersetzungsprozess für Coroutinen enthaltende Klassen leicht modifiziert werden muss, wie im folgenden Abschnitt beschrieben wird.

Bytecode-Rewriting

Eine zentrale Eigenschaft in der Sprachdefinition der Programmiersprache Java ist das Fehlen einer unbedingten Sprunganweisung. Es existiert zwar die Möglichkeit, Schleifen vorzeitig auch aus verschiedenen Verschachtelungstiefen heraus abzubrechen (*break/continue*), aber ein klassisches *goto* wurde beim Sprachentwurf bewusst weggelassen.[7]

Dennoch sind sowohl die Ablaufsteuerung von RIFE/CONTINUATIONS als auch die von JAVAFLOW Umsetzungen der in Abschnitt 4.6 beschriebenen Ablaufsteuerung durch unbedingte Sprünge. Dabei bedienen sich beide Continuation-Implementationen der gleichen Grundidee: der Verlagerung der Sprungproblematik von der Sprache Java in den vom Java-Compiler generierten Bytecode.

Zunächst stellen sich beide Continuation-Implementationen dem Nutzer als gewöhnliche Java-Bibliotheken dar, d. h., sie reichern die Sprache Java mit zusätzlichen Klassen- und Methodendefinitionen an. Wie im vorherigen Abschnitt am Beispiel von JAVAFLOW gezeigt, befindet sich unter diesen Erweiterungen insbesondere auch eine Klasse, deren Instanzen Repräsentationen von Continuations darstellen und die bereits die Definitionen zweier Methoden für das Unterbrechen und Fortsetzen von Coroutinen, *suspend* und *continue* enthält.

Der mit Hilfe dieser Klassen und Methoden entstehende Quelltext ist trotz der Existenz von Coroutinen/Continuations gültig im Sinne der Java-Spezifikation. Die Verwendungen von *suspend* und *continue* sind aus Sicht des Java-Compilers reguläre Methodenaufrufe und werden dementsprechend auch in der generierten Klassendatei als solche umgesetzt. Allerdings sollten derartige, Continuations enthaltende, Klassendateien nach der Kompilation nicht ausgeführt werden, da die Implementationen von *suspend* und *continue* leer sind, d. h., jeder Aufruf dieser vordefinierten Methoden kehrt ohne irgendeinen Effekt unmittelbar zum Rufer zurück.

[7] Das Schlüsselwort *goto* ist zwar in der Sprachspezifikation enthalten, dient jedoch dem Java-Compiler, um bei einer versehentlichen Benutzung eine bessere Fehlerausgabe zu ermöglichen [AGH06].

Wie bereits erwähnt, erfolgt die eigentliche Implementation des Coroutinen/Continuation-Verhaltens durch eine Modifikation des vom Java-Compiler generierten Bytecodes. Dazu analysiert ein separates Programm, der sogenannte Bytecode-Rewriter, die generierten Klassendateien und sucht nach Aufrufen der beiden Methoden *suspend* und *continue*. Findet er derartige Aufrufe, so modifiziert er den gesamten Methodenkörper, in dem der Aufruf enthalten war, so dass die Methode anschließend den Aufbau besitzt, der in Abschnitt 4.6 bei der Beschreibung der Ablaufsteuerung durch unbedingte Sprunganweisungen vorgestellt wurde (am Beispiel gezeigt in Quelltext 4.3 auf Seite 64).

Neben dieser strukturellen Anpassung zum Zwecke der späteren Ablaufsteuerung werden zusätzlich Quelltextpassagen eingefügt, die die Kontextsicherung durch explizites Sichern und Wiederherstellen der Coroutinen-lokalen Variablen übernehmen. Im folgenden Abschnitt werden die konkreten Realisierungen sowohl der Ablaufsteuerung als auch der Kontextsicherung an einem Beispiel demonstriert.

Das Umschreiben des Bytecodes kann wahlweise statisch oder dynamisch durchgeführt werden. Im statischen Fall handelt es sich beim Bytecode-Rewriter um ein externes Programm, das nach dem Java-Compiler aufgerufen wird, wobei die umzuschreibenden *class*-Dateien per Kommandozeilenparameter referenziert werden. In diesem Fall wird der Inhalt dieser Dateien mit dem modifizierten Bytecode überschrieben. Anschließend kann das Programm direkt von jeder beliebigen *Java Virtual Machine* (JVM), auch auf anderen Plattformen, ausgeführt werden, da die Coroutinensemantik vollständig im modifizierten Bytecode enthalten ist. Beim dynamischen Umschreiben bleiben hingegen alle *class*-Dateien auf Dauer unverändert erhalten. Stattdessen wird der Aufruf der JVM dahingehend modifiziert, dass sie einen speziellen Klassenlader (*classloader*) verwendet. Dieser untersucht während der Ausführung des Java-Programmes jede neu geladene Klasse auf potentiell enthaltene Coroutinen und modifiziert bei Vorhandensein den Bytecode wie oben beschrieben.

Die statische Variante hat den Vorteil, dass der Aufwand des Bytecode-Rewritings genau einmal auftritt, während er bei der dynamischen Variante bei jedem Programmaufruf (bei einer Simulation also in jedem einzelnen Simulationslauf) nötig ist. Dabei sollte aber erwähnt werden, dass der Prozess der Bytecode-Modifikation nur einen Bruchteil der Laufzeit auch kleiner Simulationen einnimmt. Demgegenüber hat das dynamische Bytecode-Rewriting den Vorteil, dass man sich während der Entwicklung nicht darum kümmern muss, auch wirklich alle Coroutinen-enthaltenden Klassen im Kompilationsprozess vom Bytecode-Rewriter modifizieren zu lassen.

Generell lässt sich festhalten, dass keine der beiden Varianten gravierende Vor- oder Nachteile gegenüber der anderen besitzt, so dass sich ein Benutzer relativ frei für eine von beiden entscheiden kann. Im Rahmen dieser Arbeit wurde bei der Implementation bevorzugt auf die statische zurückgegriffen.

```
1  public class MyCoroutine implements Runnable {
2      @Override
3      public void run() {
4          float a;
5          a = 1;
6          Continuation.suspend();
7          a = 2;
8          Continuation.suspend();
9          a = 3;
10     }
11 }
```

Quelltext 4.8: Java-Beispiel für die Implementation einer Coroutine in JAVAFLOW

Bytecode-Rewriting am Beispiel

Das Verfahren der Bytecode-Modifikation lässt sich am besten an einem Beispiel erklären. Da es sich beim Java-Bytecode um einen Binärcode handelt, der dementsprechend nur schwer lesbar ist, wird im Folgenden stattdessen die Assemblervariante *Jasmin* zur Darstellung verwendet. Bei Jasmin handelt es sich um eine Assemblersprache, die im Rahmen eines Buches über die *Java Virtual Machine* entstanden ist [MD97] und inzwischen den Quasi-Standard für die Assemblerrepräsentation des Java-Bytecodes einnimmt.[8]

Zur Demonstration soll eine Coroutine dienen, die in Quelltext 4.8 zu sehen ist. Diese wurde bewusst einfach gestaltet und mit sich wiederholenden Anweisungen versehen, um die Jasmin-Darstellung des daraus resultierenden Bytecodes verständlich zu halten.

In Quelltext 4.9 ist die Jasmin-Repräsentation des Bytecodes zu sehen, den der Java-Compiler bei einer Übersetzung der Coroutine aus Quelltext 4.8 erzeugt.[9] Es ist gut zu erkennen, dass die Aufrufe von *suspend* direkt als Methodenaufrufe (*invokestatic*, Zeile 7 und 10) in den Bytecode übernommen wurden. Die Zeilen 5–6, 8–9 und 11–12 enthalten die Bytecode-Umsetzung der drei Zuweisungen.

Im den Quelltexten 4.10 und 4.11 (auf den Seiten 82 und 83) ist zu sehen, wie der Bytecode der Coroutine nach der Modifikation durch den Bytecode-Rewriter aussieht. Zur leichteren Erkennung wurden alle neu hinzugekommenen Quelltextteile mit einem dunkleren Hintergrund versehen; die hell unterlegten Quelltextteile sind (von hinzugekommenen Labels abgesehen) unverändert geblieben.

Grundsätzlich hat die gesamte Methode den Aufbau erhalten, der in Abschnitt 4.6 als „Ablaufsteuerung durch unbedingte Sprünge“ vorgestellt wurde. Die Grundlage

[8] Auf eine detaillierte Einführung in diese Sprache soll an dieser Stelle verzichtet werden; die im Folgenden erscheinenden Beispiele sind aber auch ohne Vorkenntnisse von Jasmin weitestgehend verständlich. Bei Detailfragen helfen neben dem bereits genannten Buch [MD97] vor allem die Webseiten des Jasmin-Projektes [MR] und die des Nachfolgeprojektes TINAPOC [Rey] sowie die Spezifikation der Java Virtual Machine [LY99] weiter.

[9] Dieser Jasmin-Quelltext wurde, wie alle noch folgenden Jasmin-Beispiele, mit Hilfe des Java-Disassemblers CLASSFILEANALYZER (CAN) [Roe] generiert.

```
1  .method public run()V
2      .limit stack 1
3      .limit locals 2
4
5      ldc 1.0
6      fstore_1
7      invokestatic org/apache/commons/javaflow/Continuation/suspend()V
8      ldc 2.0
9      fstore_1
10     invokestatic org/apache/commons/javaflow/Continuation/suspend()V
11     ldc 3.0
12     fstore_1
13     return
14 .end method
```

Quelltext 4.9: Jasmin-Darstellung des Bytecodes der Coroutine aus Quelltext 4.8

bietet dabei eine durch die neuen Quelltextzeilen hinzugekommene (Bytecode-bedingt namenlose) Variable vom Typ Integer, die die Stelle repräsentiert, an der die Coroutine das letzte Mal unterbrochen wurde (genaugenommen werden einfach alle im Ausgangsquelltext vorkommenden *suspend*-Aufrufe durchnummeriert).

Zur Kontextsicherung wird eine Komponente namens *StackRecorder* herangezogen, die für jeden in Java vorkommenden Datentyp einen eigenen Datenstack inklusive Zugriffsmethoden (*pushInt*, *popInt*, *pushFloat*, ...) zur Verfügung stellen. Beim Betreten der umgeschriebenen Methode wird zunächst eine Referenz auf den zugehörigen *StackRecorder* geholt (Zeile 4). Sollte diese leer sein, so wurde die Coroutine noch nie gestartet und es wird direkt an den Anfang des unmodifizierten Quelltextes (Label2, Zeile 36) gesprungen. Ist die Referenz hingegen gültig, wird der oberste Wert vom Integerstack genommen (Zeile 13). Bei diesem Wert handelt es sich immer um den letzten Wert der oben beschriebenen, den letzten Unterbrechungspunkt beschreibende, Integervariablen. Abhängig von diesem Wert werden verschiedene Subroutinen angesprungen (*tableswitch* in Zeile 14–17).

In diesen so angesprungenen Programmzeilen wird abhängig vom fortzusetzenden Zustand der Kontext wiederhergestellt (Zeile 18–25 bzw. 27–33). Die Kontextwiederherstellung erfolgt dabei durch variablenweises Auslesen der Stacksicherungskomponente und Speichern der Werte in den entsprechenden Variablenplätzen. Bemerkenswert ist dabei, dass auch die *this*-Referenz mit einer gespeicherten Kopie überschrieben wird (Zeile 23–25 bzw. 32–35), um die Objektidentität über alle gespeicherten Continuations hinweg zu erhalten. Anschließend wird die der letzten Unterbrechung folgende Programmzeile direkt angesprungen (Zeile 26 bzw. 35).[10]

[10] Genaugenommen wird die letzte Unterbrechung erneut angesprungen und anschließend die im Folgenden beschriebene Kontextsicherung übersprungen. Die genaue Beschreibung dieses Mechanismus' ist jedoch für das Verständnis der Arbeitsweise nicht nötig.

```
1  .method public run()V
2      .limit stack 4
3      .limit locals 3
4      invokestatic org/apache/commons/javaflow/bytecode/StackRecorder/get()
5                           Lorg/apache/commons/javaflow/bytecode/StackRecorder;
6      dup
7      astore_2
8       ifnull  Label2
9      aload_2
10     getfield org/apache/commons/javaflow/bytecode/StackRecorder/isRestoring Z
11     ifeq Label2
12     aload_2
13     invokevirtual org/apache/commons/javaflow/bytecode/StackRecorder/popInt()I
14     tableswitch 0 1
15             Label0
16             Label1
17             default : Label2
18 Label0:
19     aload_2
20     invokevirtual org/apache/commons/javaflow/bytecode/StackRecorder/popFloat()F
21     fstore_1
22     aload_2
23     invokevirtual org/apache/commons/javaflow/bytecode/StackRecorder/popObject()Ljava/lang/Object;
24     checkcast MyCoroutine
25     astore_0
26     goto Label3
27 Label1:
28     aload_2
29     invokevirtual org/apache/commons/javaflow/bytecode/StackRecorder/popFloat()F
30     fstore_1
31     aload_2
32     invokevirtual org/apache/commons/javaflow/bytecode/StackRecorder/popObject()Ljava/lang/Object;
33     checkcast MyCoroutine
34     astore_0
35     goto Label5
36 Label2:
37     ldc 1.0
38     fstore_1
39 Label3:
40     invokestatic org/apache/commons/javaflow/Continuation/suspend()V
```

Quelltext 4.10: Jasmin-Darstellung des Bytecodes aus Quelltext 4.9 nach der Modifikation durch den JAVAFLOW-Bytecode-Rewriter (Teil 1)

```
41    aload_2
42     ifnull  Label4
43    aload_2
44     getfield  org/apache/commons/javaflow/bytecode/StackRecorder/isCapturing Z
45     ifeq  Label4
46    aload_2
47    aload_0
48     invokevirtual  org/apache/commons/javaflow/bytecode/StackRecorder/pushReference(Ljava/lang/Object;)V
49    aload_2
50    aload_0
51     invokevirtual  org/apache/commons/javaflow/bytecode/StackRecorder/pushObject(Ljava/lang/Object;)V
52    aload_2
53    fload_1
54     invokevirtual  org/apache/commons/javaflow/bytecode/StackRecorder/pushFloat(F)V
55    aload_2
56    iconst_0
57     invokevirtual  org/apache/commons/javaflow/bytecode/StackRecorder/pushInt(I)V
58    nop
59    return
60  Label4:
61     ldc  2.0
62     fstore_1
63  Label5:
64     invokestatic  org/apache/commons/javaflow/Continuation/suspend()V
65    aload_2
66     ifnull  Label6
67    aload_2
68     getfield  org/apache/commons/javaflow/bytecode/StackRecorder/isCapturing Z
69     ifeq  Label6
70    aload_2
71    aload_0
72     invokevirtual  org/apache/commons/javaflow/bytecode/StackRecorder/pushReference(Ljava/lang/Object;)V
73    aload_2
74    aload_0
75     invokevirtual  org/apache/commons/javaflow/bytecode/StackRecorder/pushObject(Ljava/lang/Object;)V
76    aload_2
77    fload_1
78     invokevirtual  org/apache/commons/javaflow/bytecode/StackRecorder/pushFloat(F)V
79    aload_2
80    iconst_1
81     invokevirtual  org/apache/commons/javaflow/bytecode/StackRecorder/pushInt(I)V
82    nop
83    return
84  Label6:
85     ldc  3.0
86     fstore_1
87     return
88  .end method
```

Quelltext 4.11: Jasmin-Darstellung des Bytecodes aus Quelltext 4.9 nach der Modifikation durch den JAVAFLOW-Bytecode-Rewriter (Teil 2)

Auf die Kontextwiederherstellungspassagen folgen die Anweisungen der eigentlichen Coroutine (Zeile 36–40 bzw. 60–64), die jeweils mit einem Aufruf von *suspend* enden. Diesen Aufrufen folgt im modifizierten Bytecode stets ein neuer Quelltextbereich, in dem die Kontextsicherung durchgeführt wird (Zeile 41–54 bzw. 65–78). Diese erfolgt analog zur Kontextwiederherstellung durch variablenweises Sichern der Werte in die Stacksicherungskomponente, wobei diesmal die Variablen in genau entgegengesetzter Reihenfolge bezüglich des späteren Auslesens durchgegangen werden. Sind alle Coroutinen-lokalen Variablen gesichert, wird der Wert der Coroutinenzustandsvariable neu gesetzt und ebenfalls in der Stacksicherungskomponente gesichert (Zeile 55–57 bzw. 79–81), bevor durch ein abschließendes *return* (Zeile 59 bzw. 83) die Coroutine wirklich unterbrochen (bzw. im Sinne der JVM beendet) wird.

Mit dem Verfahren der nachträglichen Bytecode-Modifikation und dessen Umsetzung im Webapplikationsframework JAVAFLOW konnte eine Coroutinen/Continuation-Implementation gefunden werden, die alle der in Abschnitt 4.5 zusammengetragenen Anforderungen für die Umsetzung optimistisch-paralleler Prozesse erfüllt. Daher wurde JAVAFLOW als Grundlage für die Prozessimplementation der im Rahmen dieser Arbeit entstandenen Simulationsbibliothek MYTIMEWARP gewählt.

KAPITEL 5

IMPLEMENTATION

Nachdem in den vergangenen Kapiteln die theoretischen Grundlagen sowie prinzipielle Implementationsansätze für verschiedene Teilaspekte eines optimistisch-parallelen Simulationskerns beschrieben wurden, widmet sich dieses Kapitel nun der Beschreibung der im Rahmen dieser Arbeit entstandenen Simulationsbibliothek namens MyTimeWarp.

Zunächst wird ein Überblick über die Architektur von MyTimeWarp gegeben, wobei alle Klassen kurz mit den ihnen zugewiesenen Aufgaben vorgestellt werden. Der darauffolgende, größte Teil des Kapitels widmet sich dann der Umsetzung der verschiedenen, in den Grundlagenkapiteln beschriebenen Konzepte, wobei die bereits vorgestellten Klassen detaillierter betrachtet werden. Zum Ende des Kapitels wird die Simulationsbibliothek aus Nutzersicht beschrieben und es werden die Einschränkungen in der gegenwärtigen Implementation diskutiert.

5.1 Grundarchitektur

Im Klassendiagramm in Abbildung 5.1 ist ein erster Überblick über die Architektur von MyTimeWarp zu sehen. In der Darstellung wurden alle Klassen, die den Simulationskern bilden, vor einem gemeinsamen Hintergrund zusammengefasst.

Im Zentrum einer jeden mit MyTimeWarp implementierten Simulation steht eine Instanz der Klasse *Simulation*. Diese erfüllt während eines Simulationslaufes sämtliche Aufgaben der externen Steuerung der LPs. Zu diesem Zweck besitzt sie Referenzen auf alle bei der Simulation beteiligten LPs, bei denen es sich in der MyTimeWarp-Implementation um Instanzen der Klasse *LogicalProcess* handelt. Zur serialisierten Ausgabe besitzt jeder LP eine eigene Instanz der Klasse *SerializedOutput*. Auch die zentrale *Simulation*-Instanz besitzt eine serialisierte Ausgabe, in der zum Simulationsende die Ergebnisse der serialisierten Ausgaben der einzelnen LPs zusammengefügt werden. Die Implementation des Nachrichtenaustausches zwischen

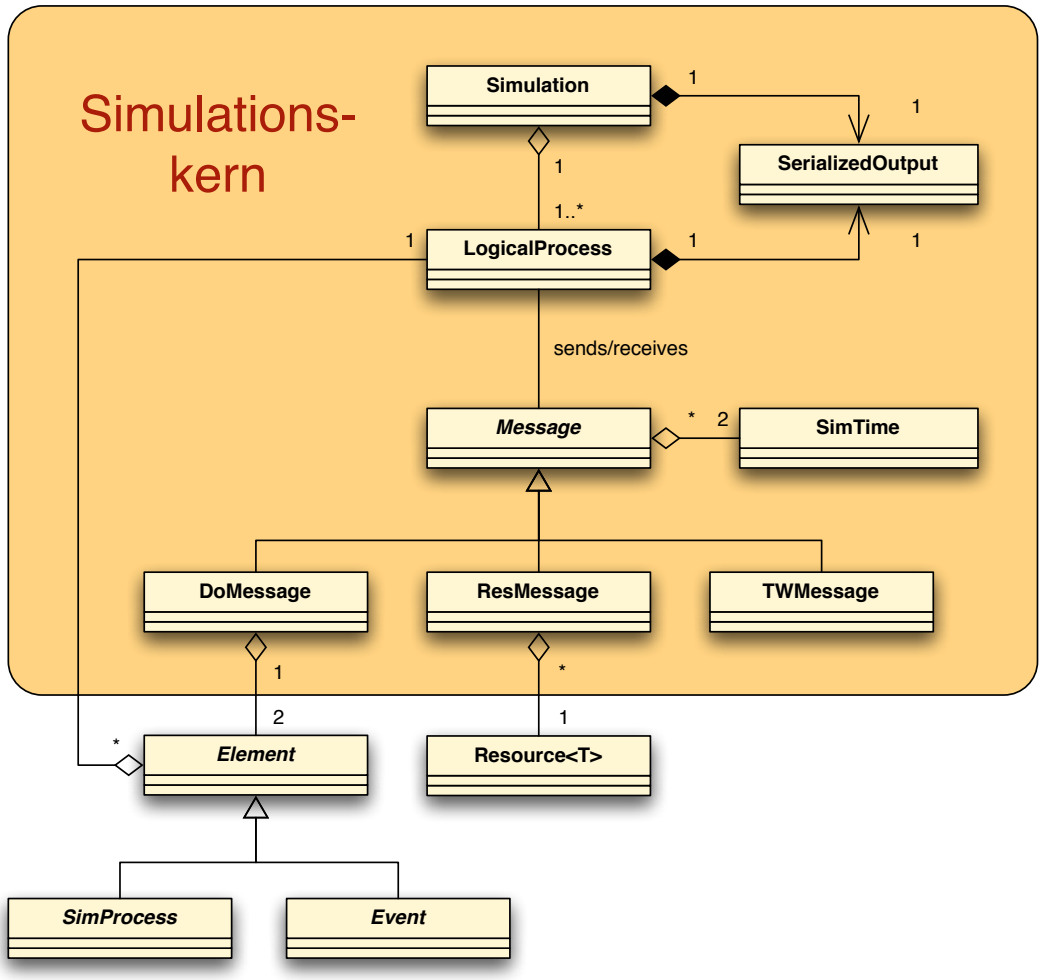

Abbildung 5.1: Überblick über die Architektur von MYTIMEWARP

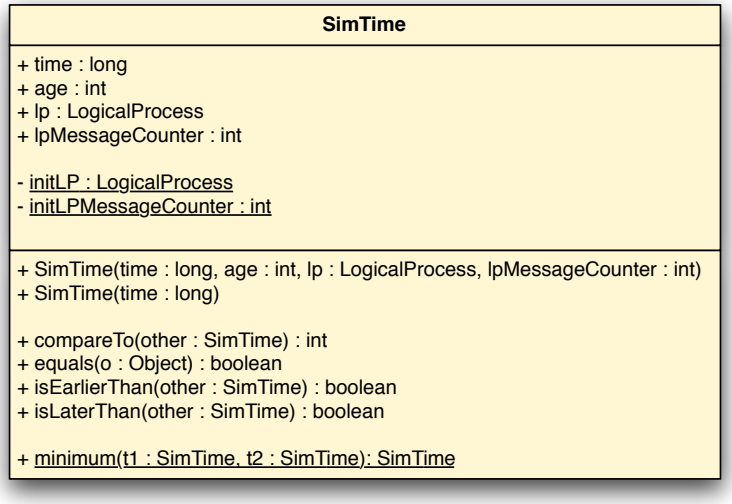

Abbildung 5.2: Die Klasse *SimTime*

LPs basiert auf Instanzen der Konkretisierungen der abstrakten Klasse *Message*, wobei je nach benötigtem Nachrichtentyp eine der Konkretisierungen *DoMessage*, *ResMessage* oder *TWMessage* gewählt wird. Alle Nachrichten enthalten Referenzen auf genau zwei Modellzeiten, welche wiederum durch Instanzen der Klasse *SimTime* umgesetzt werden.

Die restlichen abgebildeten Klassen *Element*, *SimProcess*, *Event* und *Resource* gehören nicht zum Simulationskern, sondern stellen die Grundlage für das Erstellen eigener Simulationsmodelle dar.

5.2 Realisierung der grundlegenden Simulationskonzepte

Umsetzung der Modellzeit

Auch wenn ein Nutzer von MYTIMEWARP stets nur mit skalaren Modellzeiten vom Datentyp *int* in Berührung kommt, so wird innerhalb des Simulationskerns ausschließlich mit strukturierten Modellzeiten gearbeitet. Wie in Abschnitt 2.7 detailliert beschrieben wurde, lässt sich nur dadurch sicherstellen, dass alle Simulationsergebnisse eines optimistisch-parallelen Simulationsprogrammes reproduzierbar sind.

In MYTIMEWARP sind alle strukturierten Modellzeiten Instanzen der Klasse *SimTime*, deren Klassendiagramm in Abbildung 5.2 zu sehen ist. Bei den Attributen der Klasse handelt es sich um 1:1-Umsetzungen der in Abschnitt 2.7 identifizierten Attribute einer strukturierten Modellzeit:

- die skalare Modellzeit im Attribut *time*,
- das Altersattribut im Attribut *age*,

- eine Referenz auf den LP, der die Modellzeit generiert hat, im Attribut *lp* sowie
- die Anzahl der von diesem LP bereits generierten Nachrichten im Attribut *lpMessageCounter*.

Zur Instantiierung stehen zwei Konstruktoren zur Verfügung. Der erste erfordert die Angabe aller vier Attribute als Parameter und erstellt eine entsprechende Instanz der Modellzeit. Der zweite Konstruktor, der lediglich die Angabe einer skalaren Modellzeit als Parameter erwartet, ist hingegen nur für die Erstellung von Modellzeiten initialer Ereignisse zuständig. Das Setzen des Altersattributes ist dabei unproblematisch, da dieses bei initialen Ereignissen stets den Wert 0 besitzen muss. Dafür ergibt sich ein Problem bei der Belegung des dritten und vierten Attributes (*lp* und *lpMessageCounter*), da es bei initialen Ereignissen keinen LP gibt, der die Generierung der Modellzeit veranlasst haben kann.

Die Lösung besteht in den beiden Klassenattributen *initLP* und *initLPMessageCounter*. Es wird genau einmal zu Beginn eines Simulationslaufes ein spezieller LP namens *initLP* instantiiert, dessen einzige Aufgabe darin besteht, in Modellzeiten initialer Ereignisse im *lp*-Attribut referenziert zu werden. Eine analoge Bedeutung besitzt *initLPMessageCounter*, dessen Wert bei jeder neuen initialen Modellzeit als viertes Attribut verwendet wird. Da *initLPMessageCounter* nach jeder Verwendung inkrementiert wird, ergibt sich bei einer späteren Betrachtung der initialen Modellzeiten der (gewollte) Eindruck, dass der LP *initLP* nacheinander alle initialen Ereignisse durch Erzeugen von Nachrichten generiert hat.

Umsetzung von Modellierungselementen

Eine zentrale Eigenschaft von MYTIMEWARP besteht darin, dass der Simulationskern sowohl ereignisorientierte als auch prozessorientierte Simulationsmodelle sowie Simulationsmodelle, die Elemente beider Modellierungssichten enthalten, ausführen kann.

Wie in Abbildung 5.3 zu sehen ist, sind die Grundelemente beider Modellierungssichten, Ereignisse und Prozesse, als Ableitung einer gemeinsamen Basisklasse *Element* umgesetzt worden. Diese Klasse besitzt neben einem Namensattribut eine Referenz auf einen LP, der für den Rest des Simulationslaufes für die Ausführung dieses Elementes zuständig ist. Des Weiteren enthält *Element* mehrere Methoden, auf die ein Modellierer bei der Erstellung der Ereignisroutinen bzw. Prozesslebensläufe zurückgreifen kann, wie im Folgenden gezeigt wird.

Ereignisse

Die Grundlage eines ereignisorientierten Simulationsmodells besteht in einer Menge von Ereignissen, die voneinander abhängig sind. Analog zu anderen ereignisorientierten Simulationsbibliotheken erfolgt die Umsetzung eines Ereignisses in MYTIMEWARP durch die Implementation eines Ereignistyps in der Ableitung einer abstrakten

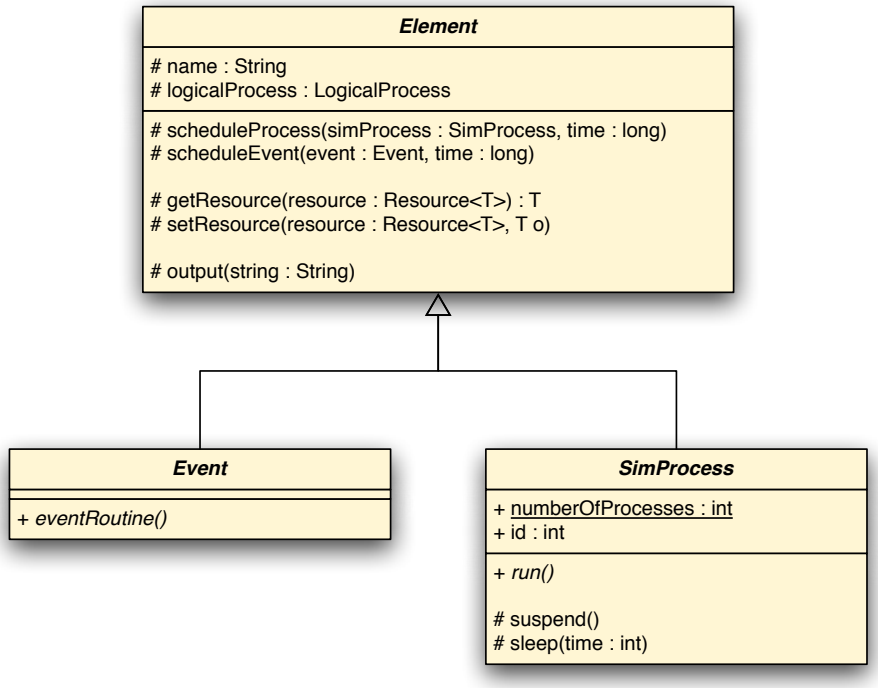

Abbildung 5.3: Die abstrakte Klasse *Element* mit ihren ebenfalls abstrakten Ableitungen *Event* und *SimProcess*

Ereignisklasse (*Event*). Dabei muss die Implementation der Ereignisroutine in der (in *Event* abstrakt deklarierten) Methode *eventRoutine* erfolgen.

Innerhalb der Ereignisroutine ist die Verwendung beliebigen Java-Quelltextes erlaubt. Des Weiteren stellt *Element*, die Mutterklasse von *Event*, eine Methode namens *scheduleEvent* zur Verfügung, mittels der neue Ereignisse im Ereigniskalender eingetragen werden können. Analog kann mit der Methode *scheduleProcess* das Starten bzw. die Fortführung eines Prozesses veranlasst werden.

Eine weitere, von *Element* angebotene Methode namens *output* kann zur Ausgabe einfacher Zeichenketten (*Strings*) verwendet werden. Eine direkte Nutzung der Standard-Ausgabe-Mechanismen von Java wie z. B. durch den Aufruf von *System.out.print* ist zwar ebenfalls möglich, allerdings zur Ausgabe des momentanen Modellzustandes ungeeignet, da bedingt durch die optimistisch-parallele Ausführung derartig getätigte Ausgaben nachträglich ungültig werden können. Alle mittels *output* getätigten Ausgaben werden hingegen solange zurückgehalten, bis ihre Gültigkeit definitiv gesichert ist.

Prozesse

Die Grundlage von prozessorientierten Simulationsmodellen bilden in MYTIMEWARP Instanzen der Klasse *SimProcess*. Dabei werden die Lebenszyklen der einzelnen Prozesse in den Körpern der jeweils zu implementierenden Methode *run* umgesetzt. Innerhalb dieser Lebenszyklusmethoden kann auf alle bereits beschriebenen Methoden der Oberklasse *Element* zugegriffen werden.

Zusätzlich bietet *SimProcess* jedoch noch zwei Methoden an, die das bisher viel diskutierte Unterbrechen eines Prozesses ermöglichen. Dabei sorgt *suspend* für eine reine Unterbrechung während *sleep* eine Unterbrechung inklusive Fortsetzung nach der im Parameter angegebenen Zeitspanne einleitet.

Nachrichten

Auch wenn aus Sicht des Simulationsmodellierers Ereignisse mittels einfacher Methoden (*scheduleEvent*, *scheduleProcess*) in einen Ereigniskalender eingetragen werden, so führt doch jeder Aufruf einer dieser Methoden zu einem Nachrichtenaustausch zwischen LPs. Selbst Ereignisse, die ein LP für seine eigene Zukunft einplant, werden in Nachrichten verpackt.

Die Grundlage aller Nachrichten bildet die abstrakte Klasse *Message*, deren Klassendiagramm in Abbildung 5.4 zu sehen ist. Neben einer eindeutigen Identifikationsnummer besitzt jede Nachricht zwei Modellzeiten: die Modellzeit des sendenden LPs zum Zeitpunkt des Sendens sowie die Modellzeit, zu der der Nachrichteninhalt bearbeitet werden soll, was je nach Typ der Nachricht unterschiedliche Bedeutungen haben kann.

Wie in Abbildung 5.4 zu sehen ist, existieren insgesamt drei konkrete Ableitungen von *Message*. Die Nachrichten vom Typ *DoMessage* werden von LPs benutzt, um sich gegenseitig über Ereignisse zu informieren. Dazu verfügen derartige Nachrichten

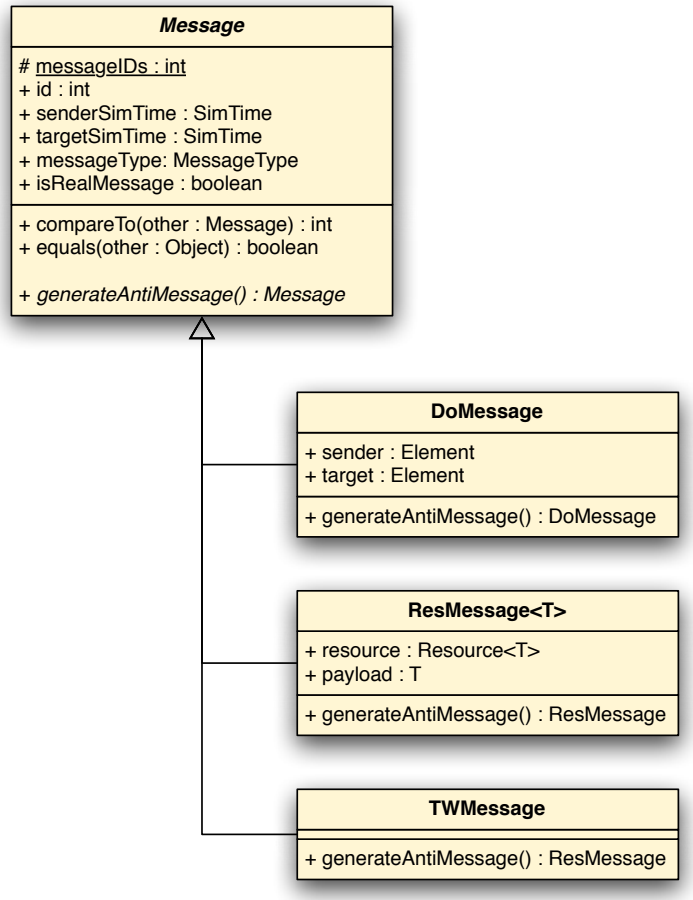

Abbildung 5.4: Die abstrakte Klasse *Message* mit ihren Konkretisierungen

über zwei weitere Attribute, die das sendende Element sowie das in der Zukunft auszuführende Element referenzieren. Die Nachrichten vom Typ *ResMessage* und *TWMessage* werden hingegen ausschließlich zur Kommunikation zwischen LPs und Ressourcen benötigt. Eine ausführliche Beschreibung befindet sich in Abschnitt 5.4 bei der Vorstellung der Ressourcen-Implementation.

Ein boolesches Attribut namens *isRealMessage* in allen Nachrichten legt fest, ob es sich bei der jeweiligen Nachricht um eine „echte“ Nachricht oder aber um eine Anti-Nachricht handelt. Zur Erstellung von Anti-Nachrichten existiert in *Message* die abstrakte Methode *generateAntiMessage*, die in jeder konkreten Ableitung adäquat zum jeweiligen Nachrichtentyp implementiert worden ist.

5.3 Umsetzung der logischen Prozesse

Struktureller Aufbau

Wie in Kapitel 3 beschrieben wurde, besteht die Grundlage jeder optimistisch-parallelen Simulation in einer Menge von LPs, die unabhängig voneinander die ihnen zugeordneten Ereignisse abarbeiten. In MYTIMEWARP wurde diese parallele Arbeitsweise von LPs dadurch umgesetzt, dass jeder LP in einem separaten Java-Thread abläuft. Die Implementation der LPs erfolgte dabei in der Klasse *LogicalProcess*, die das Java-Interface *Runnable* implementiert.

Ein Nebeneffekt dieser Implementationsstrategie besteht darin, dass die letztendliche Zuordnung von LPs auf Recheneinheiten dynamisch durch die JVM durchgeführt wird. Gleichzeitig wird damit die Problematik gelöst, die sich ergibt, wenn mehr LPs existieren als Recheneinheiten zur Verfügung stehen. In diesem Fall übernimmt die JVM gleichzeitig die Aufgabe der Lastverteilung auf die verfügbaren Recheneinheiten, also z. B. die Bündelung wenig aktiver LPs auf einer Recheneinheit, damit sehr aktiven LPs andere Recheneinheiten exklusiv zustehen können. Dank der ständigen Neuzuordnung der Java-Threads auf Recheneinheiten bereiten dabei auch starke Sprünge in der Aktivität der LPs keine Probleme.

In Abbildung 5.5 ist ein Klassendiagramm der Klasse *LogicalProcess* zu sehen. Zunächst besitzt jeder LP eine eindeutige Identifikationsnummer (*LPid*), eine Referenz auf das zentrale Simulationsobjekt (*simulation*) sowie eine eigene serialisierte Ausgabe (*localSerializedOutput*). Zur Nachrichtenverwaltung besitzt jeder LP fünf Nachrichtenpuffer, in denen Nachrichten, nach ihrem Ausführungszeitpunkt sortiert, vorgehalten werden können. In Abbildung 5.6 sind schematisch die Nachrichtenpuffer zweier LP zu sehen. Die Pfeile symbolisieren dabei die unterschiedlichen Wege, auf denen sich Nachrichten bzw. Anti-Nachrichten zwischen den Puffern bewegen können. Dabei stellen die dickeren Pfeile den regulären Ablauf einer Simulation dar, während Nachrichten entlang der dünneren Pfeile nur bei Time-Warps auftreten. Des Weiteren stehen vollständige Linien für Wege von „echten“ Nachrichten, während gestrichelte Linien Wege von Anti-Nachrichten symbolisieren.

Abbildung 5.5: Die Klasse *LogicalProcess*

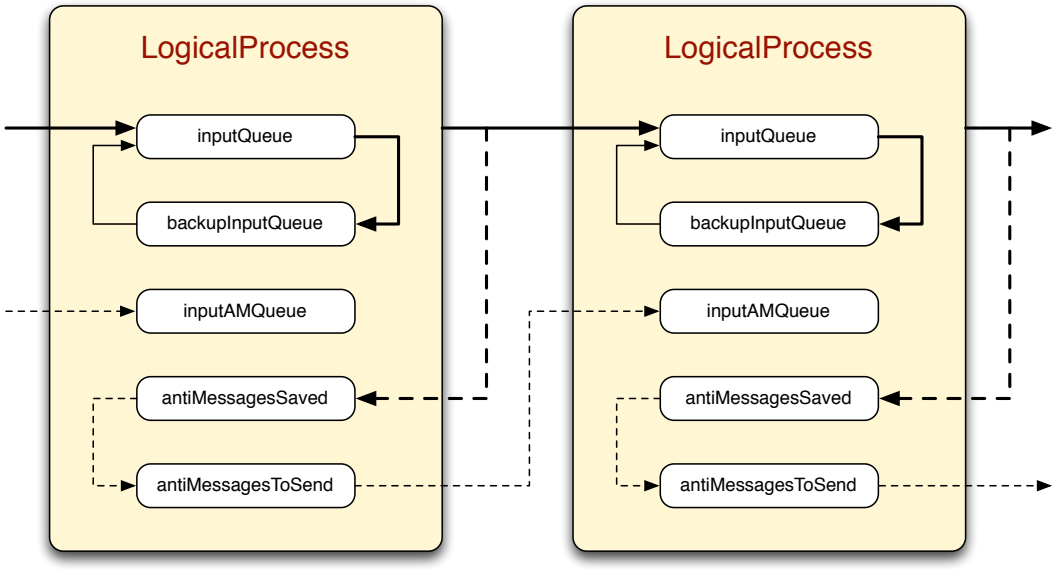

Abbildung 5.6: Überblick über die Nachrichtenkanäle in und zwischen LPs

Wann immer ein LP einen anderen über ein neues Ereignis informiert, wird eine Nachricht vom ersten zum zweiten verschickt, die letztgenannter im Puffer *inputQueue* speichert. Gleichzeitig erstellt der sendende LP eine der Nachricht zugehörige Anti-Nachricht und speichert diese in seinem eigenen Puffer *antiMessagesSaved*. Die Bearbeitung eines Ereignisses durch einen LP führt dazu, dass die das Ereignis enthaltende Nachricht aus *inputQueue* nach *backupInputQueue* verschoben wird.

Wenn nun in einem LP ein Time-Warp ausgelöst wird, so werden nacheinander folgende Aktionen ausgeführt:

- Alle bereits bearbeiteten Nachrichten, die Ereignisse enthalten, deren Eintrittszeitpunkte nach dem Time-Warp wieder in der Zukunft des LPs liegen werden aus *backupInputQueue* nach *inputQueue* verschoben.
- Alle Anti-Nachrichten zu bereits verschickten, aber durch den Time-Warp nun ungültigen Nachrichten werden von *antiMessagesSaved* nach *antiMessagesToSend* verschoben.
- Im weiteren Verlauf (im nächsten unsynchronisierten Block, siehe weiter unten) werden alle Nachrichten aus *antiMessagesToSend* an die LPs verschickt, die die ursprünglichen Nachrichten erhalten haben. Die empfangenden LPs speichern dabei alle Anti-Nachrichten, die sie nicht sofort bearbeiten können (z. B. wenn dazu ein Time-Warp nötig ist) im Puffer *inputAMQueue*.

Speziell für die Bearbeitung von Prozessen besitzt jeder LP zwei weitere Datenstrukturen in Form von Hashtabellen: *processStackMap* und *processHeapMap*. Erstere speichert während eines Simulationslaufes zu jedem Prozess, der von diesem LP bearbeitet wird, eine sortierte Liste. In dieser wird wiederum zu jedem Modellzeit-

punkt, zu dem der Prozess aktiv war, der Laufzeitzustand des Prozesses nach der Ausführung in Form einer Continuation hinterlegt.

Die Datenstruktur *processHeapMap* ist äquivalent aufgebaut, nur dass hier statt Laufzeitzuständen Kopien der Prozessobjekte zu allen aktiven Modellzeiten hinterlegt werden. Dadurch werden auch die Belegungen der Prozessvariablen außerhalb der Lebenszyklusmethode zwischen zwei Aktivierungen desselben Prozesses konserviert. Diese Datenstruktur *processHeapMap* wird benötigt, da die verwendete Continuation-Implementation JAVAFLOW keine Objektvariablen sichert (siehe Abschnitt 4.7) und ansonsten Prozesse keine Prozessvariablen besitzen dürften.

Die restlichen Attribute werden in der folgenden Beschreibung der Arbeitsweise eines LPs vorgestellt.

Arbeitsweise

Die Theorie optimistisch-paralleler Simulationen geht von einer völlig asynchronen Durchführung der einzelnen Teilaufgaben der LPs aus. Damit gehen jedoch größere Synchronisationsprobleme beim verteilten Zugriff auf gemeinsame Variablen einher.

Besonders problematisch ist dabei der Fall, in dem während der Bearbeitung eines Ereignisses selbiges durch das Eintreffen einer neuen Nachricht ungültig wird. Rein theoretisch müsste nun die Abarbeitung des Ereignisses sofort gestoppt und ein Time-Warp eingeleitet werden. Dabei müssten allerdings auch die durchgeführten Operationen (Änderungen an Zustandsvariablen, verschickte Nachrichten, ...) dieser letzten, partiell ausgeführten Ereignisroutine einzeln identifiziert und rückgängig gemacht werden. Diese aufwendige Aufgabe kann umgangen werden, wenn Ereignisroutinen bzw. Prozesslebenslaufabschnitte atomar betrachtet werden, d. h., beim Eintreffen einer Time-Warp-auslösenden Nachricht wird selbst ein ungültiges Ereignis noch bis zum Ende bearbeitet und erst dann wird der Time-Warp durchgeführt.

In MYTIMEWARP wurde daher der Ansatz gewählt, bestimmte Methoden bzw. Teilbereiche von Methoden so zu gestalten, das diese nur exklusiv (im Java-Sprachgebrauch: *synchronisiert*) ausgeführt werden können. Als Mittel zur Umsetzung wurde das bereits in Abschnitt 4.6 ausführlich erklärte Monitor-Konzept von Java verwendet. Das Synchronisationsobjekt ist dabei in allen Fällen das Objekt des konkreten LPs selbst (referenziert durch *this*).

Da der konkrete Javacode der Implementation der LPs nicht nur durch die Synchronisation vergleichsweise komplex ist, wird im Folgenden die Arbeitsweise nur stark verkürzt und bei Bedarf in Pseudocode dargestellt. Dabei werden in der Darstellung alle Passagen, die nur synchronisiert ausgeführt werden können, mit einem dunklen Hintergrund versehen. Wann immer *irgendeine* Methode eines LPs einen derartigen Bereich betreten hat, müssen *alle* anderen Methoden mit dem Betreten ihrer geschützten Bereiche warten, bis die erste Methode ihren wieder verlässt.

Der Versand von Nachrichten, die über einzuplanende Ereignisse informieren sollen, erfolgt durch den Aufruf der unsynchronisierten Methode *sendDoMessage* im sendenden LP. Diese Methode erstellt ein Nachrichtenobjekt und ruft dann am emp-

```
1  public synchronized void receiveMessage(Message message) {
2
3      if (message.targetSimTime.isLaterThan(currentTime)) {
4          if (message.isRealMessage) {
5              inputQueue.add(message);
6          }
7          else {
8              inputQueue.remove(message);
9          }
10     }
11     else {
12         timeWarpOccurred = true;
13
14         if (message.isRealMessage) {
15             backupInputQueue.add(message);
16         }
17         else {
18             inputAMQueue.add(message);
19         }
20     }
21
22     simulation.reportActiveLP(this);
23     notify();
24 }
```

Quelltext 5.1: Die Methode *receiveMessage* eines LPs (gekürzt)

fangenden LP die synchronisierte Methode *receiveMessage*, die stark verkürzt in Quelltext 5.1 zu sehen ist. In *receiveMessage* wird zunächst überprüft, ob der Eintrittszeitpunkt des in der Nachricht enthaltenen Ereignisses in der Zukunft liegt (Zeile 3). Im Positivfall erfolgt bei einer Nachricht die Einordnung selbiger in den Eingangspuffer *inputQueue* (Zeile 4–6), bei einer Anti-Nachricht hingegen wird die zugehörige Nachricht aus dem Eingangspuffer entfernt (Zeile 7–9). In beiden Fällen ist kein Time-Warp notwendig.

Liegt hingegen der Eintrittszeitpunkt des Ereignisses der eingegangenen Nachricht in der Vergangenheit, so wird zunächst eine Variable namens *timeWarpOccurred* gesetzt (Zeile 12). Wie bei der Beschreibung der Methode *run* noch zu sehen sein wird, sorgt dies dafür, dass der LP bei nächster Gelegenheit einen Time-Warp durchführt. Im Falle einer Nachricht wird diese dann direkt der Liste bereits bearbeiteter Nachrichten *backupInputQueue* hinzugefügt (Zeile 14–16); der Time-Warp wird später dafür sorgen, dass die Nachricht in den Eingangspuffer *inputQueue* verschoben wird. Handelte es sich hingegen um eine Anti-Nachricht, so wird diese im Puffer *inputAMQueue* gespeichert.

Schließlich wird das zentrale Simulationsobjekt darüber informiert, dass der aktuelle LP noch bzw. wieder aktiv ist und am LP die Methode *notify* gerufen. Diese sorgt dafür, dass der LP reaktiviert wird, falls er sich bereits im Leerlauf befand.

Dadurch, dass *receiveMessage* vollständig synchronisiert ist, wird zum einen sichergestellt, dass der gleichzeitige Empfang mehrerer Nachrichten keine Probleme bereitet (der zweite Aufruf von *receiveMessage* wird solange verzögert, bis der erste beendet ist) und zum anderen, dass auch keine andere Methode des LPs auf *inputQueue* zugreift (was insbesondere im Fall eines gerade durchgeführten Time-Warps fatal wäre).

Die Implementation der zentralen, über dem Eingangspuffer iterierenden Schleife erfolgt in der Methode *run*. In Quelltext 5.2 ist eine Pseudocode-Darstellung dieser Methode zu sehen. Zu Beginn jeder Iteration wird in einem synchronisierten Block überprüft, ob seit dem letzten Schleifendurchlauf eine Nachricht eingegangen ist, deren Bearbeitung einen Time-Warp erfordert. Falls ja, wird zunächst dieser Time-Warp durchgeführt, indem die Nachrichten wie oben beschrieben zwischen den Puffern verschoben werden (Zeile 4–9) und alle nun ungültigen Prozesszustandsinformationen entsorgt werden (Zeile 10–11). Anschließend wird die früheste Nachricht aus dem Eingangspuffer entfernt und in *currentMessage* gespeichert.

Wie bereits beschrieben, wird die Bearbeitung aller eintreffenden Nachrichten während der Bearbeitung dieses synchronisierten Blocks blockiert. Sobald der LP den nun folgenden unsynchronisierten Block betritt, werden die eingetroffenen Nachrichten parallel zu den beschriebenen Operationen ausgewertet. In diesem unsynchronisierten Block werden zunächst alle bei der Bearbeitung des letzten Time-Warps zum Versenden identifizierten Anti-Nachrichten verschickt. Das Verschicken der Anti-Nachrichten muss zwingend asynchron erfolgen, da sich ansonsten zwei LPs gegenseitig in eine Deadlock-Situation bringen können (jeder LP wartet in seinem synchronisierten Block darauf, dass der andere seinen verlässt). Anschließend wird die Methode *evaluateCurrentMessage* aufgerufen, die sich um die eigentliche Abarbeitung des in der aktuellen Nachricht enthaltenen Ereignisses kümmert.

Es folgt wieder ein synchronisierter Block (Zeile 20–27), in dem geprüft wird, ob noch weitere unbearbeitete Nachrichten vorliegen oder aber parallel zur Bearbeitung des letzten Ereignisses eine Nachzüglernachricht eingetroffen ist. Im Positivfall fährt die Schleife mit der nächsten Iteration fort, im Negativfall geht der LP nach der Benachrichtigung des zentralen Simulationsobjektes in einen Wartezustand über.

Dieser Wartezustand kann nur auf zwei Wegen beendet werden. Zum einen, wenn eine neue Nachricht eintrifft (siehe die Beschreibung von *receiveMessage*) und zum anderen, wenn das zentrale Simulationsobjekt feststellt, dass sich alle LPs im Leerlauf befinden. In letzterem Fall setzt das Simulationsobjekt bei allen LPs die Variable *killSwitch* und sorgt dann bei jedem LP für dessen Fortsetzung mittels *notify*. Da jedoch die zentrale Schleife jedes LPs von der initialen Belegung der Variable *killSwitch* mit *false* abhängig ist, werden dadurch alle LPs zur Beendigung ihrer Methode *run* gebracht.

Die eigentliche Auswertung einer Nachricht erfolgt in der bereits erwähnten Methode *evaluateCurrentMessage*, die in Quelltext 5.3 dargestellt ist. Gleich zu Beginn erfolgt die Unterscheidung nach dem Typ des in der Nachricht enthaltenen Elementes (Zeile 2, 6 und 37).

```
1  public void run() {
2      while (! killSwitch ) {
3          if (timeWarpOccured) {
4              entferne alle Nachrichten aus backupInputQueue, deren Anti−Nachrichten in
5                      inputAMQueue enthalten sind
6              verschiebe alle (wieder aktuellen) Nachrichten aus backupInputQueue
7                      nach inputQueue
8              verschiebe Anti−Nachrichten zu bereits versendeten, aber nun ungültigen
9                      Nachrichten aus antiMessagesSaved nach antiMessagesToSend
10             entferne alle Prozessinformationen über nun ungültige Prozesszustände aus
11                     processStackMap und processHeapMap
12             timeWarpOccured = false;
13         }
14
15         if (!inputQueue.isEmpty()) {
16             currentMessage = inputQueue.first();
17         }
18         verschicke alle Nachrichten aus antiMessagesToSend
19         evaluateCurrentMessage();
20         if (! killSwitch && inputQueue.isEmpty() && !timeWarpOccurred) {
21             teile Simulation die Deaktivierung dieses LPs mit
22             if (! killSwitch ) wait ();
23
24             if (! killSwitch ) {
25                 teile Simulation die Reaktivierung dieses LPs mit
26             }
27         }
28     }
29     füge serialisierte Ausgabe des LPs der serialisierten Ausgabe der Simulation hinzu
30 }
```

Quelltext 5.2: Die Methode *run* eines LPs (Pseudocode-Darstellung)

```
1  private void evaluateCurrentMessage() {
2      if (currentMessage.target instanceof Event) {
3          ((Event) currentMessage.target).eventRoutine();
4      }
5
6      else if (currentMessage.target instanceof SimProcess) {
7          SimProcess currentProcess = (SimProcess) currentMessage.target;
8          Continuation newState;
9          SortedMap<SimTime, Continuation> newStackMap;
10         SortedMap<SimTime, SimProcess> newHeapMap;
11
12         if (!processStackMap.containsKey(currentProcess)
13                 || processStackMap.get(currentProcess).isEmpty()) {
14
15             newStackMap = new TreeMap<SimTime, Continuation>();
16             newHeapMap = new TreeMap<SimTime, SimProcess>();
17             processStackMap.put(currentProcess, newStackMap);
18             processHeapMap.put(currentProcess, newHeapMap);
19             currentProcess = currentProcess.clone();
20
21             newState = Continuation.startWith(currentProcess, currentProcess);
22         }
23
24         else {
25             newStackMap = processStackMap.get(currentProcess);
26             newHeapMap = processHeapMap.get(currentProcess);
27             currentProcess = newHeapMap.get(newHeapMap.lastKey()).clone();
28
29             newState = Continuation.continueWith(
30                 newStackMap.get(newStackMap.lastKey()), currentProcess
31             );
32         }
33
34         newStackMap.put(currentTime, newState);
35         newHeapMap.put(currentTime, currentProcess);
36     }
37     else {
38         logger. fatal (" invalid  message type in inputQueue");
39     }
40 }
```

Quelltext 5.3: Die Methode *evaluateCurrentMessage* eines LPs (gekürzt)

Die Abarbeitung eines Ereignisses ist sehr einfach, da lediglich die Ereignisroutine des Ereignisses direkt aufgerufen werden muss (Zeile 3, siehe auch Abschnitt 2.8). Die Bearbeitung einer Prozessaktivierung ist erwartungsgemäß wesentlich komplizierter (siehe Abschnitt 2.8). Zunächst wird unterschieden (Zeile 12–13), ob es sich um eine erstmalige Aktivierung oder aber um die Fortsetzung eines Prozesses handelt.

Im Fall der ersten Aktivierung (Zeile 15–21) werden zunächst neue assoziative Listen für die Continuations *newStackMap* und Objektvariablen *newHeapMap* dieses neuen Prozesses angelegt. Diese werden anschließend in den bereits beschriebenen Datenstrukturen *processStackMap* und *processHeapMap* eingetragen, wobei das Prozessobjekt selbst als Schlüssel dient.

Schließlich wird der Prozess gestartet (Zeile 21). Sobald die Ausführung des Prozesses (vorläufig) beendet ist, wird die zurückgegebene Continuation in *newState* gespeichert.

Der Quelltext für die Fortsetzung eines Prozesses (Zeile 25–35) ist dem für die erstmalige Aktivierung ähnlich. Ein wesentlicher Unterschied besteht darin, dass bei einer Fortsetzung *newStackMap* und *newHeapMap* aus *processStackMap* und *processHeapMap* besorgt werden. Anschließend wird die letzte gespeicherte Continuation aus *newStackMap* unter Verwendung des letzten Objektvariablensatzes aus *newHeapMap* fortgesetzt.

Das Klonen des Objektvariablensatzes in den Zeilen 19 und 27 ist jeweils nötig, da dieser durch die Ausführung des Prozesses verändert werden kann. Da im Falle eines Time-Warps jedoch die jeweils vorher gültigen Belegungen der Objektvariablen wiederhergestellt werden müssen, wird stets eine Kopie in der Fortsetzung verwendet.

Unabhängig davon, ob ein Prozess zum ersten Mal gestartet oder fortgesetzt wurde, werden der neue Laufzeitzustand und der neue Objektvariablensatz in *newStackMap* bzw. *newHeapMap* gespeichert (Zeile 34–35).

Bei dem schon mehrfach erwähnten zentralen Simulationsobjekt handelt es sich um eine Instanz der Klasse *Simulation*, deren Klassendiagramm in Abbildung 5.7 zu sehen ist. Dieses Objekt erfüllt während eines Simulationslaufes drei Aufgaben. Es ist dafür zuständig:

- zum Simulationsbeginn jeden LP in einem separaten Thread zu starten,
- während einer Simulation darüber zu wachen, welche LPs noch aktiv sind und im Falle eines Leerlaufes aller LPs die Beendigung selber einzuleiten und
- zum Simulationsende die lokalen serialisierten Ausgaben der LPs zusammenzufassen.

Diesen Aufgaben entsprechend besitzt das Simulationsobjekt eine Liste von Threads (*threads*), je eine Liste von existierenden und aktuell arbeitenden LPs (*existingLPs* und *runningLPs*) sowie eine eigene serialisierte Ausgabe (*serializedOutput*).

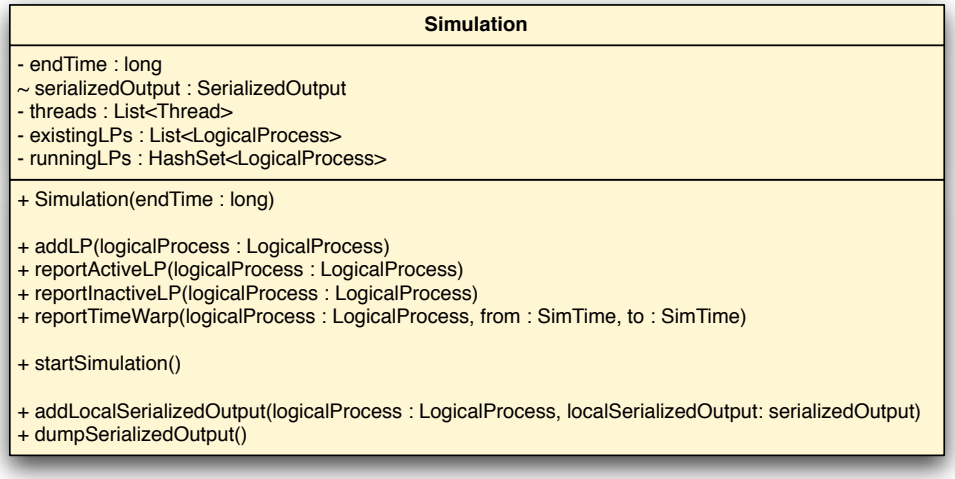

Abbildung 5.7: Die Klasse *Simulation*

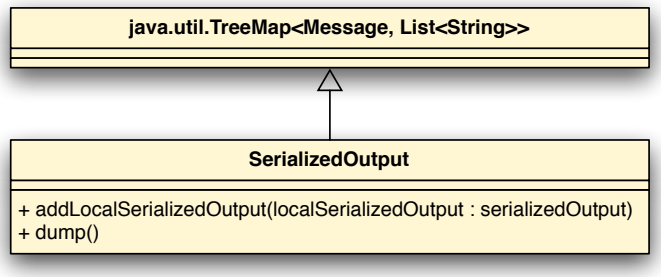

Abbildung 5.8: Die Klasse *SerializedOutput*

5.4 Realisierung weiterer Konzepte

Serialisierte Ausgabe

Bei den serialisierten Ausgaben der LPs und des zentralen Simulationsobjektes handelt es sich um Instanzen der Klasse *SerializedOutput*, die im Klassendiagramm in Abbildung 5.8 dargestellt ist. Jede serialisierte Ausgabe stellt eine sortierte assoziative Liste dar (umgesetzt durch eine Ableitung von der entsprechenden Java-Implementation einer derartigen Liste in der Klasse *TreeMap*). In diesen Listen werden Nachrichten als Schlüsselwerte und verkettete Listen von Zeichenketten als Datenwerte verwendet.

Wenn ein LP das erste Mal bei der Bearbeitung eines Ereignisses eine Zeichenkette ausgeben soll, so wird in der serialisierten Ausgabe ein neues Wertepaar angelegt, wobei als Schlüssel die das Ereignis enthaltende Nachricht verwendet wird. Als Datenwert dient eine neue Liste von Zeichenketten, deren erstes und vorerst einziges

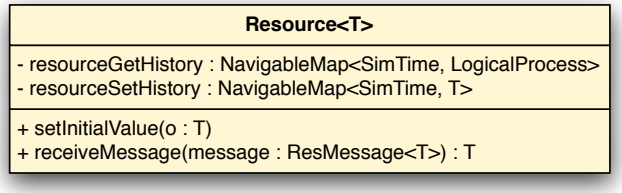

Abbildung 5.9: Die Klasse *Resource*

Element die auszugebende Zeichenkette darstellt. Soll während der Abarbeitung dieses Ereignisses eine weitere Zeichenkette ausgegeben werden, so wird diese der Liste von Zeichenketten hinzugefügt. Bei jeder Ausgabe im Rahmen der Bearbeitung eines weiteren Ereignisses wird hingegen wieder ein neues Wertepaar von Nachricht und Liste von Zeichenketten angelegt.

Ressourcen

Die separate Implementation des Ressourcen-Konzeptes lässt sich durchaus als eine Art Komfortmerkmal auffassen, da dessen Existenz zwar die Modellierung deutlich vereinfacht, es aber prinzipiell auch möglich wäre, Ressourcen in Form von Prozessen zu implementieren. Um Ressourcen zu ermöglichen, die sich weitestgehend wie Ressourcen aus sequentiellen Simulationsbibliotheken verwenden lassen, war es nötig, die Implementation der LPs sowie des Nachrichtenaustausches zu erweitern. Die Implementation der Ressourcen selbst erfolgte in der parametrisierten Klasse *Resource*, deren Klassendiagramm in Abbildung 5.9 zu sehen ist.

Die Grundidee hinter den implementierten Ressourcen besteht darin, dass jede Ressource sich in zwei separaten Listen merkt, wann sie von welchem LP gelesen (*resourceGetHistory*) und wann sie mit welchem Wert beschrieben (*resourceSetHistory*) wurde. Dadurch ist sie prinzipiell in der Lage, anfragenden LPs zu unterschiedlichen Modellzeiten die zu diesen Zeitpunkten jeweils gültigen Werte mitzuteilen.

Wann immer ein LP eine Ressource lesen oder schreiben möchte, so ruft er an dieser die Methode *receiveMessage* und übergibt dabei eine Instanz der Klasse *ResMessage* (Klassendiagramm in Abbildung 5.4 auf Seite 91). Ob es sich um einen Lese- oder Schreibzugriff handelt, wird dabei in einem Attribut dieser Nachricht (*messageType*) festgelegt. Im Falle eines Schreibzugriffs enthält das Attribut *payload* den neuen Wert der Ressource, bei einem Lesezugriff liefert die Methode den Wert der Ressource als Rückgabewert.

Die konkrete Bearbeitung einer eingehenden Nachricht hängt sowohl von der in der Nachricht enthaltenen, aktuellen Modellzeit des sendenen LPs (*senderSimTime*) als auch von den bereits vorher empfangenen Nachrichten ab. Lesezugriffe sind prinzipiell unproblematisch. Die Ressource sucht in diesem Fall den letzten Schreibzugriff, der in einer Modellzeit vor der des lesenden LPs liegt, und gibt den mit diesem

Schreibzugriff verbundenen Wert zurück. Anschließend wird dieser Lesezugriff in der Liste *resourceGetHistory* vermerkt.

Auch das Zurückziehen eines Lesezugriffs bereitet keine Probleme. Sollte zu einer vergangenen Leseoperation eine Anti-Nachricht (also eine Anti-Nachricht vom Typ *ResMessage* eingehen), weil der ehemals lesende LP einen Time-Warp durchgeführt hat, so wird einfach der vermerkte Lesezugriff aus *resourceGetHistory* gestrichen.

Die Behandlung von Schreibzugriffen ist etwas komplizierter. Zunächst wird der Schreibzugriff in *resourceSetHistory* vermerkt. Anschließend wird geprüft, ob (in der Modellzeit) nach diesem Schreibzugriff bereits ein Lesezugriff erfolgte. Falls ja, so wird der zugehörige LP durch den Versand einer Nachricht vom Typ *TWMessage* (Klassendiagramm in Abbildung 5.4 auf Seite 91) zu einem Time-Warp zurück zum Zeitpunkt des Lesezugriffes gezwungen. Der Time-Warp dieses LPs wird dann als Nebeneffekt dafür sorgen, dass der ungültige Lesezugriff auch aus der Liste *resourceGetHistory* verschwindet.

Ähnlich erfolgt die Bearbeitung eines zurückgezogenen Schreibzugriffes. Nachdem der ursprüngliche Schreibzugriff aus *resourceSetHistory* entfernt wurde, werden alle LPs zum Time-Warp gezwungen, die (in der Modellzeit) nach dem entfernten Schreibzugriff auf die Ressource lesend zugegriffen haben.

Dem Endanwender bleiben nicht nur die Bearbeitung von Lese- und Schreibzugriffen, sondern auch der gesamte Versand der Nachrichten vom Typ *ResMessage* verborgen. Dank der zwei Methoden *getResource* und *setResource* in der Klasse *Element* kann er sowohl in den Ereignisroutinen von Ereignissen als auch in den Lebenszyklusmethoden von Prozessen unter der lediglichen Angabe des Namens der Ressource auf selbige lesend und schreibend zugreifen.

5.5 MyTimeWarp aus Nutzersicht

Prinzipiell erfolgt die Implementation eines Simulationsmodells für MyTimeWarp analog zur Implementation eines Simulationsmodells für einen sequentiellen Simulationskern. Aus Nutzersicht existieren nur eine skalare Modellzeit und ein Ereigniskalender; die konsequente Verwendung einer strukturierten Modellzeit und der Versand von Nachrichten im Simulationskern bleiben dem Endanwender nahezu vollständig verborgen. Selbst Zugriffe auf Ressourcen erscheinen dem Endanwender wie einfache Variablenzugriffe.

Eine erste Besonderheit besteht allerdings bei der Initialisierung der Simulation. Im Gegensatz zu einer sequentiellen Simulation muss bei MyTimeWarp zuerst eine Menge von LPs erstellt werden. Anschließend ist der Modellierer verpflichtet, zu jedem Modellelement (also Ereignissen und Prozessen) festzulegen, auf welchem LP dieses Element die gesamte Simulation über ausgeführt werden soll. Wie noch in Kapitel 6 anhand durchgeführter Experimente demonstriert wird, hängt von dieser Verteilung von Elementen auf LPs maßgeblich die zu erreichende Simulationsbeschleunigung durch Parallelisierung ab.

Eine weitere Einschränkung vor allem gegenüber sequentiellen prozessorientierten Simulationsbibliotheken besteht darin, dass alle Prozessinstanzen bereits zu Simulationsbeginn instantiiert werden müssen und nicht erst im Laufe einer Simulation erstellt können. In vielen Simulationsmodellen lässt sich diese Einschränkung dadurch umgehen, dass alle erst später entstehenden Prozesse dennoch zu Simulationsbeginn erstellt und umgehend bis zu ihrem eigentlichen Start deaktiviert werden.

Dass das Erstellen einer Simulation mit MYTIMEWARP dennoch einfach ist, sollen unter anderem die implementierten Simulationsmodelle in Anhang A demonstrieren. Die implementierten Beispiele sind dort vollständig dargestellt, d. h. insbesondere, dass in keinem Beispiel weitere als die gezeigten Quelltexte benötigt werden.

5.6 Einschränkungen von MYTIMEWARP

Obwohl sich mit MYTIMEWARP durchaus schon recht komplexe Simulationsmodelle implementieren lassen, so befindet sich die Simulationsbibliothek dennoch in einem Zustand, der sich am ehesten als *proof-of-concept* bezeichnen lässt.

Zwei der größten Mankos von MYTIMEWARP sind die derzeit nicht vorhandenen Implementationen von GVT-Berechnung und Freigabe obsoleter Prozesszustände (fossil collection). Auch wenn bei den Experimenten im Rahmen dieser Arbeit kein Simulationslauf mehr als den auf der jeweils verwendeten Hardware vorhandenen Hauptspeicher benötigte, so stellt diese potentielle Einschränkung doch eine ernsthafte Hürde für einen produktiven Einsatz von MYTIMEWARP dar.

Des Weiteren fehlen MYTIMEWARP Hilfsmittel zur Erfassung und Auswertung von statistischen Daten. Im aktuellen Zustand von MYTIMEWARP besteht die einzige Möglichkeit in der Ausgabe von Zeichenketten, die mit der jeweils aktuellen Modellzeit verknüpft sind. Auf dieser Basis sind detaillierte Auswertungen eines Simulationslaufes zwar möglich, aber sehr aufwendig. Daher stellt die Implementation von statistischen Hilfsmitteln, vor allem bei einer angedachten Anwendung von MYTIMEWARP in der Praxis, ebenfalls eine dringende Aufgabe dar.

KAPITEL 6

EXPERIMENTE

Das Ziel bei den im Rahmen dieser Arbeit durchgeführten Simulationsläufen bestand in Gegensatz zu denen üblicher Simulationsexperimente nicht in der Ermittlung von Zustandsvariablenbelegungen, die bzw. deren Entwicklung sich als Simulationsergebnisse interpretieren lassen. Genaugenommen ist das verwendete Simulationsszenario sogar hinreichend einfach, so dass sich dessen Entwicklung auch analytisch erschließen lässt. Dieser Umstand wurde ausgenutzt, um die Korrektheit der einzelnen Simulationsläufe zu überprüfen.

Stattdessen bestand die primäre zu ermittelnde Größe in jedem Simulationslauf in der (Ausführungs-)Zeitspanne, die für die vollständige Ausführung des Simulationsmodells benötigt wurde, um so ein Maß für die Beschleunigung durch die parallele Simulationsausführung zu erhalten. Im Folgenden werden zunächst das Basisszenario sowie das zu dessen Berechnung gewählte Simulationsmodell inklusive Parameter vorgestellt. Anschließend wird ein Überblick über die durchgeführten Experimente gegeben, bevor diese im Einzelnen beschrieben und die dabei gemessenen Ergebnisse analysiert werden.

6.1 Ausgangsszenario und Simulationsmodell

Als zugrundeliegendes Szenario wurde eine Variation des Problems der dinierenden Philosophen (*dining philosophers*) von Dijkstra [Dij71] gewählt.

Gegeben sei ein runder Tisch, an dem n Philosophen vor jeweils einem Teller mit Nahrung sitzen. Des Weiteren sind auf dem Tisch n Gabeln verteilt, und zwar so, dass sich zwischen je zwei benachbarten Philosophen jeweils genau eine befindet.

Das Verhaltensmuster aller Philosophen ist identisch: sobald ein Philosoph zu existieren beginnt, versucht er nacheinander seine linke und seine rechte Gabel aufzunehmen. Sobald er im Besitz beider Gabeln ist, beginnt er mit dem Vorgang der Nahrungsaufnahme, der die konstante Zeit t_0 dauert. Sobald er mit Essen fertig ist,

legt er beide Gabeln, beginnend mit der rechten, nacheinander ab und verfällt in einen Zustand des Denkens, der ebenfalls die Zeit t_0 beansprucht.

Sollte hingegen beim initialen Versuch der Gabelaufnahme eine der beiden Gabeln nicht verfügbar gewesen sein, so verfällt der Philosoph direkt in den Denkzustand der Zeitdauer t_0. Sollte es ihm bereits gelungen sein, in den Besitz der linken Gabel zu gelangen, so legt er diese vorher zurück.

Unabhängig davon, wie ein Philosoph in den Denkzustand gelangt ist, beginnt nach der Zeit von t_0 sein Verhalten von vorn mit der versuchten Gabelaufnahme.

Modellierung

Das beschriebene Szenario wurde prozessorientiert in ein Simulationsmodell überführt. Dabei werden die Philosophen im Modell durch Prozesse und die Gabeln durch Ressourcen vom Typ *boolean* (Gabel verfügbar bzw. nicht verfügbar) repräsentiert. Die ungekürzten und ausführlich kommentierten Quelltexte des Simulationsmodells und der zugehörigen Simulationsinitialisierung befinden sich in Anhang A.3.

Die Lebenszyklusmethode eines Philosophen ist, auf die wesentlichen Anweisungen gekürzt, in Quelltext 6.1 zu sehen. Man kann gut erkennen, dass das im obigen Szenario beschriebene Verhalten 1:1 in der Implementation umgesetzt wurde.

Den Vorgang des eigentlichen Wartens übernimmt die Methode *spendTime*, die in Quelltext 6.2 dargestellt ist. Bei einer reinen Umsetzung des beschriebenen Szenarios müsste der Körper dieser Methode eigentlich nur aus der *sleep*-Anweisung in Zeile 32 bestehen. Zu Testzwecken wurde die Methode *spendTime* jedoch zusätzlich mit einer künstlichen Verzögerung ausgestattet. Diese Verzögerung besteht wahlweise aus einer absichtlich ineffizienten, rekursiven Berechnung der m-ten Fibonaccizahl oder aber der Berechnung von m MD5-Summen. Durch diese künstliche Verzögerung ist es möglich, den Rechenaufwand eines Simulationslaufes ohne strukturelle Veränderungen im Simulationsmodell schrittweise zu erhöhen. Dadurch ist es wiederum möglich zu überprüfen, ab welcher Komplexität der anfallenden Berechnungen die durch die Parallelisierung erreichte Beschleunigung den parallelisierungsbedingten Zusatzaufwand übersteigt.

Im Folgenden werden alle Parameter vorgestellt, die vor einem Simulationslauf mit Werten belegt werden müssen. Dabei werden die Parameter getrennt nach Simulationsparametern und Laufzeitparametern (siehe Abschnitt 2.8) vorgestellt. Die verwendeten Parameternamen entsprechen den Variablennamen in der Implementation.

Simulationsparameter

In der Implementation werden folgende Simulationsparameter benutzt:

numberOfPhilosophers gibt die Anzahl der Philosophen und damit auch gleichzeitig die Anzahl der Gabeln an.

```
1  @Override
2  public void run() {
3
4      while (true) {
5          // take left
6          if (getRessource(leftFork)) {
7              setRessource(leftFork, false);
8              output(name + " ... takes left");
9
10             // take right
11             if (getRessource(rightFork)) {
12                 setRessource(rightFork, false);
13                 output(name + " ... takes right");
14
15                 // eating ...
16                 output(name + " ... eating");
17                 spendTime();
18
19                 // put right
20                 setRessource(rightFork, true);
21                 output(name + " ... puts right");
22             }
23
24             // put left
25             setRessource(leftFork, true);
26             output(name + " ... puts left");
27         }
28
29         output(name + " ... thinking");
30         spendTime();
31     }
32 }
```

Quelltext 6.1: Die Methode *run* eines Philosophenprozesses

```
1  protected void spendTime() {
2      if (delayParameter > 0) {
3          if (kindOfDelay == 0) {
4              // künstliche Verzögerung durch rekursive Berechnung
5              // der m−ten Fibonacci−Zahl (m = delayParameter)
6
7              dummyInt = fibo(delayParameter);
8          }
9          else if (kindOfDelay == 1) {
10             // künstliche Verzögerung durch n−fache Berechnung eines
11             // MD5−Hashes (m = delayParameter)
12
13             String string = name;
14             try {
15                 MessageDigest md = MessageDigest.getInstance("MD5");
16                 md.update(string.getBytes());
17                 hash = md.digest();
18                 for (int i=1; i<delayParameter; i++) {
19                     md.update(hash);
20                     hash = md.digest();
21                 }
22             }
23             catch (Exception e) { logger.fatal(e); System.exit(−1); }
24         }
25         else {
26             logger.fatal("unknown kind of delay requested");
27             System.exit(−1);
28         }
29     }
30
31     // Warten im Sinne der Fortführung der Modellzeit
32     sleep(10);
33 }
34
35 protected int fibo(int n) {
36     if (n < 2) return 1;
37     else return fibo(n−1)+fibo(n−2);
38 }
```

Quelltext 6.2: Die Methode *spendTime* eines Philosophenprozesses

endTime gibt die Modellzeit an, bei deren Erreichen ein Simulationslauf endet.

Die in der Szenariobeschreibung erwähnte Zeitdauer t_0, die ein Philosoph zum Essen oder Denken benötigt, ist in der Implementation unveränderlich auf den Wert 10 festgelegt worden.

Laufzeitparameter

In der Implementation werden folgende Laufzeitparameter verwendet:

kindOfDelay gibt die Art der künstlichen Verzögerung an. Eine 0 steht für die Verzögerung durch die rekursive Berechung von Fibonaccizahlen, eine 1 steht für die Berechnung von MD5-Summen.

delayParameter parametrisiert die gewählte künstliche Verzögerung. Im Falle der Fibonaccizahlenberechnung gibt dieser Verzögerungsparameter an, die wievielte Fibonaccizahl bei jedem Aufruf von *spendTime* berechnet werden soll. Bei der Berechnung von MD5-Summen bestimmt *delayParameter* die Anzahl der MD5-Summen, die pro Aufruf von *spendTime* berechnet werden sollen.

numberOfLPs gibt die Anzahl der LPs an, die zur Ausführung von Prozessen zur Verfügung stehen.

roundRobin gibt die Art und Weise an, in der die Philosophenprozesse auf die LPs verteilt werden.

Dabei stehen zwei konkrete Verteilungen zur Auswahl: die Kreissegment- und die Round-Robin-Verteilung. Bei der Round-Robin-Verteilung wird reihum jedem Philosophen der jeweils nächste LP zugewiesen, wobei bei Erreichen des letzten LPs wieder mit dem ersten fortgefahren wird. Die Round-Robin-Verteilung wird in der Implementation dadurch umgesetzt, dass dem i-ten Philosophen jeweils der LP mit der folgenden Nummer zugewiesen wird:

$$i \bmod \mathit{numberOfLPs}$$

Bei der Kreissegment-Verteilung wird hingegen zuerst der Tisch in *numberOfLPs* Kreissegmente gleicher Größe unterteilt und anschließend werden die Prozesse aller Philosophen, die am gleichen Kreissegment sitzen, demselben LP zugewiesen. In der Implementation wird eine solche Verteilung umgesetzt, indem dem i-ten Philosophen jeweils der LPs mit der folgenden Nummer zugewiesen wird:

$$\frac{i - i \bmod \left(\frac{\mathit{numberOfPhilosophers}}{\mathit{numberOfLPs}}\right)}{\frac{\mathit{numberOfPhilosophers}}{\mathit{numberOfLPs}}}$$

In Abbildung 6.1 sind sowohl die Round-Robin- als auch die Kreissegment-Verteilung schematisch für 12 Philosophenprozesse und 1 bis 4 LPs dargestellt. Dabei

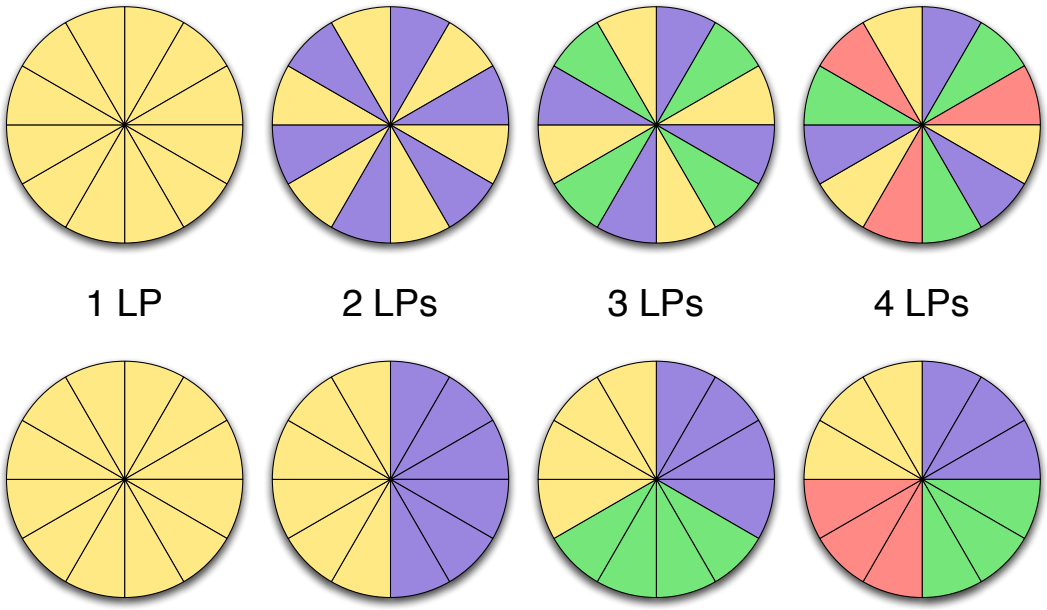

Abbildung 6.1: Round-Robin-Verteilung (oben) und Kreissegment-Verteilung (unten) am Beispiel von 12 Philosophenprozessen und 1–4 LPs

stehen die 12 Segmente pro Kreis für die einzelnen Philosophenprozesse und die 4 Farben für die LPs, denen der jeweilige Philosophenprozess zugewiesen wurde.

6.2 Überblick über die durchgeführten Experimente

Auf Basis des vorgestellten Simulationsmodells bzw. dessen Implementation mit Hilfe von MYTIMEWARP wurden zahlreiche Experimente mit verschiedenen Parameterbelegungen durchgeführt. In den folgenden Abschnitten werden die Ergebnisse von drei Experimentserien detailliert ausgewertet:

- ***32 Philosophen mit Fibonaccizahlenberechnung***
  Das Ziel dieser ersten Experimentserie war die Ermittlung grundsätzlicher Aussagen bezüglich der Abhängigkeit der Simulationslaufzeit sowohl von der Komplexität der Simulationsmodellberechnung als auch der Wahl der konkreten Verteilung von Prozessen auf LPs.

  Des Weiteren wurden die Experimente auf verschiedenen Rechnern wiederholt, um die Unabhängigkeit der ermittelten Zusammenhänge von der konkret verwendeten Hardware und dem jeweils eingesetzten Betriebssystem zu überprüfen.

- ***1024 Philosophen mit Fibonaccizahlenberechnung***
  Durch die zweite Experimentserie sollten die Ergebnisse der ersten anhand

eines leicht modifizierten Modells verifiziert werden. Es wurden daher in den Experimenten sowohl die bereits ermittelten Zusammenhänge als auch deren Unabhängigkeit von eingesetzter Hardware und Betriebssystem überprüft.

- ***32 Philosophen mit MD5-Summenberechnung***
  Das erste Ziel der dritten Experimentserie war die Validierung der Ergebnisse der ersten beiden Serien bei Verwendung einer anderen Methode der künstlichen Verzögerung. Daher wurden einmal mehr die beschriebenen Untersuchungen durchgeführt.

  Zusätzlich wurden jedoch die Experimente dieser Experimentserie auch mit einer äquivalenten, separaten Simulationsimplementation auf Basis von DESMO-J durchgeführt. So konnte eine erste Überprüfung der Leistungsfähigkeit von MYTIMEWARP im Vergleich zu einer existierenden, sequentiellen Simulationsbibliothek durchgeführt werden.

6.3 Experimentserie: 32 Philosophen mit Fibonaccizahlenberechnung

Die erste untersuchte Fragestellung bezog sich direkt auf die beabsichtigte Beschleunigung von Simulationsläufen durch die Parallelisierung der Simulationsmodellausführung. Zur Untersuchung wurde das oben beschriebene Szenario des Philosophenproblems mit Fibonaccizahlenberechnung wiederholt auf vier Rechnern (*olymp*, *mnemosyne*, *gruenau1* und *gruenau2*) ausgeführt, die sich paarweise bezüglich Hardware oder Betriebssystem unterscheiden. Eine nähere Beschreibung aller verwendeten Rechner befindet sich in Anhang D.

Parameterbelegungen

Die Simulationsparameter wurden bei allen durchgeführten Experimenten mit den gleichen Werten belegt: Das Philosophenproblem wurde für den Modellzeitraum von 0–1000 Zeiteinheiten auf der Basis von 32 Philosophen berechnet. Dabei wurde bei der Festlegung der Anzahl von Philosophen darauf geachtet, dass eine gleichmäßige Verteilung der Philosophenprozesse auf 1, 2, 4 und 8 LPs möglich ist.

Die Laufzeitparameter wurden hingegen variiert, um die verschiedenen Abhängigkeiten der für einen Simulationslauf benötigten Zeit zu ermitteln. Um die Abhängigkeit von der Anzahl verfügbarer LPs zu untersuchen, wurden für diesen Laufzeitparameter die bereits genannten Werte 1, 2, 4 und 8 eingesetzt. Der maximale Wert wurde so gewählt, dass die Anzahl der verfügbaren physischen Prozessoren bei allen verwendeten Rechnern größer ist.[1] Dadurch wurde sichergestellt, dass alle LPs prinzipiell echt parallel arbeiten können.

[1] 10 exklusiv verfügbare Prozessoren auf *olymp*; 16 potentiell mit anderen Nutzern geteilte Prozessoren auf *mnemosyne*, *gruenau1* und *gruenau2*

Um die Abhängigkeit der Simulationsbeschleunigung vom Aufwand der Modellberechnung zu untersuchen, wurde der Verzögerungsparameter im Intervall $[0, 27]$ variiert. Der maximale Wert wurde durch die benötigte Laufzeit bestimmt – Simulationsläufe mit höheren Verzögerungsparameterbelegungen benötigten zu viel Ausführungszeit, um sie hinreichend oft wiederholen zu können.

Die Abhängigkeit der benötigten Laufzeit von der konkreten Verteilung von Prozessen auf LPs wurde untersucht, indem sowohl Experimente mit der Round-Robin- als auch Experimente auf Basis der Kreissegment-Verteilung durchgeführt wurden.

Gemessene Laufzeiten und Time-Warps

Das beschriebene Simulationsmodell wurde mit den genannten Parameterbelegungen zunächst wiederholt auf *olymp* ausgeführt. In den Abbildungen 6.2 und 6.3 sind die Durchschnittswerte der dabei gemessenen Laufzeiten in Abhängigkeit von der Belegung des Verzögerungsparameters dargestellt. Um die Kurven optisch besser zu trennen, wurde die Zeitachse in beiden Diagrammen logarithmisch skaliert. Dabei stellt Abbildung 6.2 die Ergebnisse der Experimente mit Round-Robin-Verteilung dar, während die Ergebnisse in Abbildung 6.3 auf den Experimenten mit Hilfe der Kreissegment-Verteilung basieren. Die jeweils vier abgetragenen Kurven gruppieren die ermittelten Messwerte nach jeweils gleicher Anzahl der verwendeten LPs (1, 2, 4 und 8).

Bei dem verwendeten Simulationsmodell ergibt sich jede Gesamtlaufzeit eines Simulationslaufes als Summe der beiden Laufzeiten für die Simulationsmodell- und die Fibonaccizahlenberechnung.[2] Während jedoch der Aufwand für die Simulationsmodellberechnung bei allen Simulationsläufen gleich groß ist, steigt der Aufwand der Fibonaccizahlenberechnung mit zunehmendem Verzögerungsparameter exponentiell an.

Die in den beiden Diagrammen dargestellten Kurvenscharen durchlaufen alle mit steigender Verzögerungsparameterbelegung nacheinander drei Phasen. In der ersten Phase (Verzögerungsparameterbelegung 0 bis 11) erfolgt die Berechnung der Fibonaccizahlen so schnell, dass die Gesamtlaufzeit weitestgehend durch die Simulationsmodellberechnung bestimmt wird. Da letztere bei gleicher Anzahl von LPs stets konstant ist, gleichen die Kurven weitestgehend Waagerechten. Bemerkenswert ist die Tatsache, dass die für die Simulationsläufe benötigte Laufzeit mit steigender Anzahl von LPs zunimmt. Der Grund für dieses Phänomen liegt in der mit zunehmender Anzahl von LPs steigenden Häufigkeit von auftretenden Time-Warps. Der damit jeweils verbundene zusätzliche Aufwand, insbesondere die mehrfache Berechnung von Simulationsmodellzuständen eliminiert in dieser ersten Phase vollständig den Geschwindigkeitsgewinn durch die Parallelisierung.

Die dritte Phase (Verzögerungsparameterbelegung 22 bis 27) zeichnet sich hingegen durch eine Dominanz der Fibonaccizahlenberechnung aus. Mit zunehmender

[2] Auf die zusätzlich von der JVM und dem Betriebssystem beanspruchte Laufzeit soll an dieser Stelle nicht eingegangen werden, insbesondere da diese im Vergleich zu den beiden betrachteten Laufzeiten vernachlässigbar gering ist.

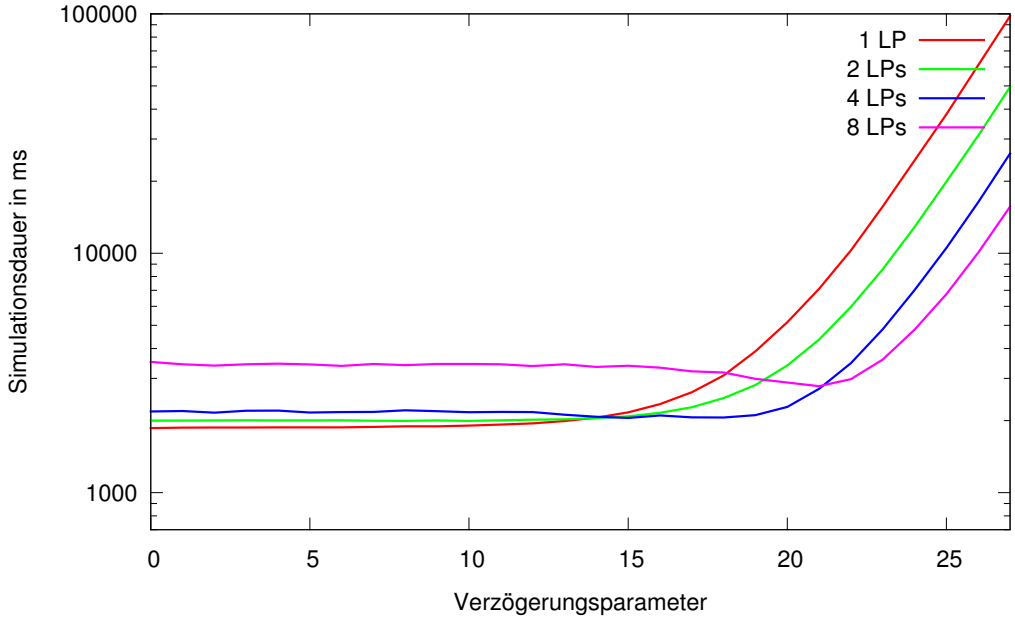

Abbildung 6.2: Gemessene Laufzeiten (32 Philosophen, Fibonaccizahlenberechnung, Round-Robin-Verteilung)

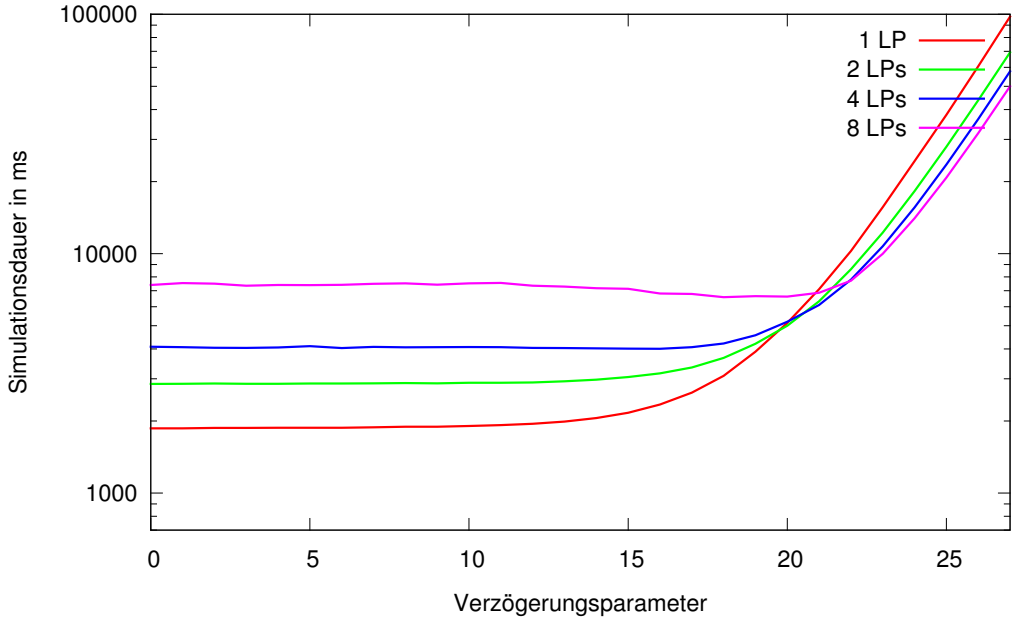

Abbildung 6.3: Gemessene Laufzeiten (32 Philosophen, Fibonaccizahlenberechnung, Kreissegment-Verteilung)

Verzögerungsparameterbelegung steigt die Gesamtlaufzeit exponentiell an, was genau dem typischen Laufzeitverhalten der rekursiven Berechnung von Fibonaccizahlen entspricht. Das exponentielle Wachstum lässt sich besonders gut daran erkennen, dass die Kurven bedingt durch die logarithmische Zeitskala die Form parallel verlaufender Geraden annehmen. Des Weiteren führt jeder zusätzliche LP zu einer Beschleunigung der Simulation, was bei einer rein parallelen Berechnung von Fibonaccizahlen erwartet wurde.

Am interessantesten ist jedoch die zweite Phase (Verzögerungsparameterbelegung 11 bis 22). Wie bereits beschrieben, steigt mit zunehmender Verzögerungsparameterbelegung die Auswirkung der parallelen Fibonaccizahlenberechnung gegenüber dem zusätzlichen Aufwand für die Time-Warp-Behandlung. In den Diagrammen ist dies besonders gut daran zu erkennen, dass sich die Reihenfolge der Kurven am linken und rechten Rand der Phase vollständig umkehrt.

Erklärungsbedürftig ist allerdings die Tatsache, warum die Kurven mit höherer LP-Anzahl vor dem Übergang in das exponentielle Wachstum eine „Delle“ haben. Diese bedeutet, dass es bei einer konstanten Anzahl von LPs einen Bereich von Verzögerungsparameterbelegungen gibt, in denen ein von der Fibonaccizahlenberechnung her aufwendigerer Simulationslauf schneller durchgeführt werden kann, als ein mit weniger Rechenaufwand verbundener. Die Ursache hierfür liegt darin, dass die Anzahl der auftretenden Time-Warps nicht nur von der Anzahl verwendeter LPs, sondern auch von der Verzögerungsparameterbelegung abhängig ist, was sich sehr gut in den Diagrammen in den Abbildungen 6.4 und 6.5 erkennen lässt. Auch in diesen Diagrammen, die die Anzahl der aufgetretenen Time-Warps in Abhängigkeit von der Verzögerungsparameterbelegung und der Anzahl der verwendeten LPs zeigen, sind die drei beschriebenen Phasen erkennbar. In der ersten Phase ist die Anzahl der Time-Warps bei gleicher Anzahl verwendeter LPs relativ konstant. Ab der zweiten Phase, in der die Fibonaccizahlenberechnung erstmals als zusätzlicher Aufwand spürbar in Erscheinung tritt, sinkt die Anzahl der auftretenden Time-Warps, bis sie in der dritten Phase auf niedrigem Niveau stagniert.

Der Grund für dieses Phänomen ist, dass der steigende Aufwand der Fibonaccizahlenberechnung während eines Simulationslaufes zu einer Harmonisierung der Modellzeiten der einzelnen LPs führt. Solange der zusätzliche Aufwand nicht vorhanden oder gering ist, kann ein LP durchaus weit in die Zukunft (bzgl. Modellzeit) „seiner“ Philosophen rechnen, bevor er von einem LP eines benachbarten Philosophen via Time-Warp zurückgesetzt wird. Dies gilt insbesondere zu Beginn eines Simulationslaufes, wenn die einzelnen LPs prinzipbedingt einzeln nacheinander gestartet werden.

Der Time-Warp ist erforderlich, da der vorangeschrittene LP bei seinen Berechnungen stets von einer konstanten Verfügbarkeit bzw. Nichtverfügbarkeit aller referenzierter Ressourcen (Gabeln) ausgegangen ist. Diese Annahme stellt sich jedoch bei der ersten Änderung an einer dieser Ressourcen durch einen anderen LP als falsch heraus, so dass der erste LP zurücksetzen und alle bereits durchgeführten Berechnungen erneut ausführen muss. Dabei ist die Wahrscheinlichkeit für das Auftreten einer Ressourcenänderung und demzufolge eines Time-Warps umso größer, je

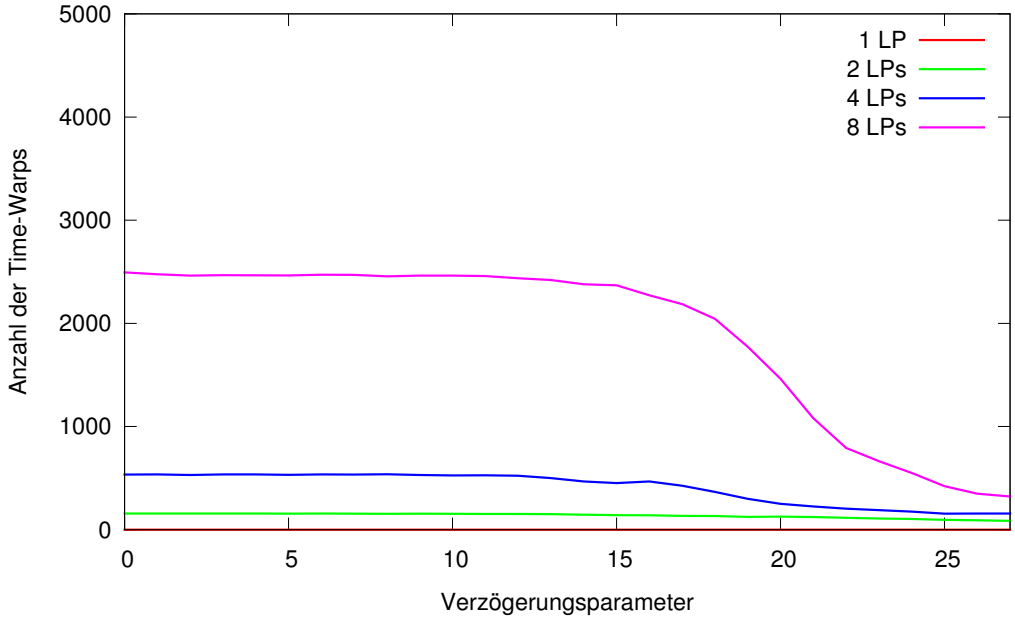

Abbildung 6.4: Gemessene Time-Warps (32 Philosophen, Fibonaccizahlenberechnung, Round-Robin-Verteilung)

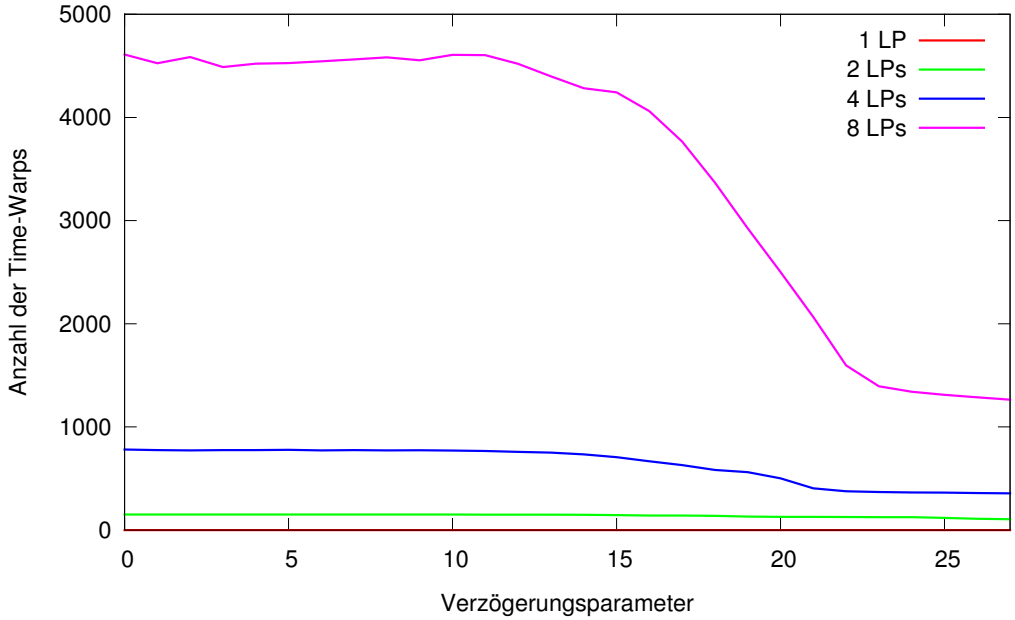

Abbildung 6.5: Gemessene Time-Warps (32 Philosophen, Fibonaccizahlenberechnung, Kreissegment-Verteilung)

weiter der LP bereits gegenüber anderen LPs in die Modellzukunft vorangeschritten ist.

Diesem wirkt eine aufwendige Berechnung von Philosophenprozessen entgegen. Bei einer zunehmenden künstlichen Verzögerung wird es für jeden einzelnen LP zunehmend schwieriger, sich beim Voranschreiten in der Modellzeit von den anderen LPs abzusetzen. In der Folge sinkt auch die Anzahl der auftretenden Time-Warps, so dass sich das in den Diagrammen gezeigte Verhalten ergibt.

Vergleich von Round-Robin- und Kreissegment-Verteilung

Im Vergleich der Abbildungen 6.2 und 6.3 lässt sich entnehmen, dass sich die gemessenen Ergebnisse der Round-Robin- und der Kreissegment-Verteilung qualitativ nicht unterscheiden. Das beschriebene Verhalten ist bei beiden Verteilungen dasselbe. Quantitativ ist jedoch den beiden, identisch skalierten Diagrammen zu entnehmen, dass bei den durchgeführten Experimenten die Round-Robin-Verteilung stets die günstigere Verteilung darstellte. Um diese Tatsache noch etwas deutlicher zu visualisieren, wurden die gemessenen Laufzeiten beider Verteilungen in Abbildung 6.6 zusammengefasst. Während die Laufzeiten bei Verwendung eines einzelnen LPs bei beiden Verteilungen erwartungsgemäß identisch sind, ergibt sich bei allen anderen gemessenen Laufzeiten ein Vorteil der Round-Robin-Verteilung gegenüber der Kreissegment-Verteilung.

Die Ursache hierfür liegt wieder in der Anzahl der aufgetretenen Time-Warps. Bei Verwendung mehrerer LPs war für jede Verzögerungsparameterbelegung die Anzahl der aufgetretenen Time-Warps bei der Kreissegment-Verteilung höher als bei der Round-Robin-Verteilung. Dies lässt sich gut den Diagrammen in den Abbildungen 6.4 und 6.5 sowie dem Diagramm in Abbildung 6.7, das die beiden vorher genannten zusammenfasst, entnehmen.

Der Grund für die unterschiedliche Anzahl der aufgetretenen Time-Warps liegt wiederum in der Anzahl von Ressourcen (Gabeln), die von Philosophenprozessen unterschiedlicher LPs geteilt werden. Wie gut anhand des Beispiels aus Abbildung 6.1 auf Seite 110 nachvollziehbar ist, werden bei der Round-Robin-Verteilung alle Ressourcen von je zwei unterschiedlichen LPs geteilt, während bei der Kreissegment-Verteilung die Anzahl derartiger Ressourcen gleich der Anzahl beteiligter LPs ist.

Je mehr derartig geteilte Ressourcen existieren, umso höher ist die Wahrscheinlichkeit, dass ein Time-Warp bei einem der beteiligten LPs auftritt. Dies klingt zunächst wie ein Argument gegen das beobachtete Verhalten. Die höhere Time-Warp-Wahrscheinlichkeit sorgt jedoch dafür, dass „irrlaufende“ LPs frühzeitig zurückgesetzt werden. Dieser Umstand führt in diesem speziellen Simulationsmodell, das sich durch eine sehr enge Kopplung der Modellelemente auszeichnet, bei der Round-Robin-Verteilung dazu, dass die Modellzeiten der einzelnen LPs eher gemeinsam wachsen. Dadurch wiederum wird jedoch die Häufigkeit der Notwendigkeit von Time-Warps stark gesenkt.

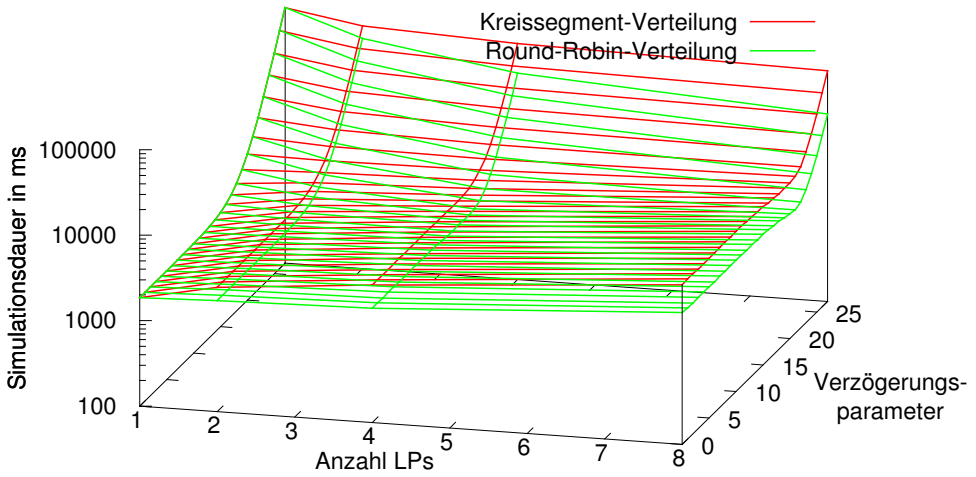

Abbildung 6.6: Gemessene Laufzeiten (32 Philosophen, Fibonaccizahlenberechnung, Round-Robin- und Kreissegment-Verteilung)

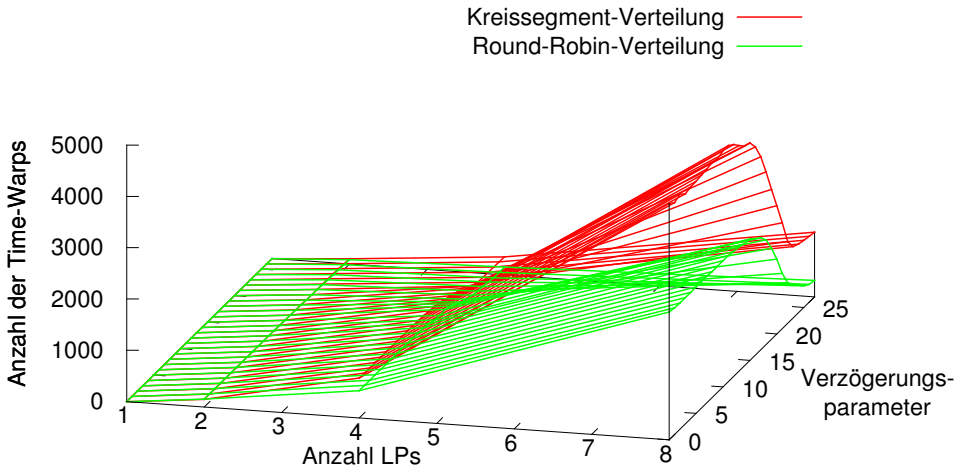

Abbildung 6.7: Anzahl der Time-Warps (32 Philosophen, Fibonaccizahlenberechnung, Round-Robin- und Kreissegment-Verteilung)

Im Gegensatz dazu weichen bei der Kreissegment-Verteilung die Modellzeiten der einzelnen LPs während eines Simulationslaufes deutlich stärker voneinander ab, was im Endeffekt zu häufiger auftretenden Time-Warps inklusive deren aufwendiger Behandlung führt.

Vergleich der Ergebnisse von verschiedenen Rechnern

Um zu bestätigen, dass alle diskutierten Phänomene unabhängig von der konkret verwendeten Kombination aus Hardware und Betriebssystem auftreten, wurden alle Experimente zusätzlich auf den Rechnern *mnemosyne*, *gruenau1* und *gruenau2* wiederholt. In Abbildung 6.8 sind die einander entsprechenden Diagramme der gemessenen Laufzeiten bzw. der aufgetretenen Time-Warps auf den drei genannten Rechnern sowie *olymp* dargestellt. Dabei ist die Skalierung der Achsen bei allen äquivalenten Diagrammen die gleiche.

Wie man gut erkennen kann, ist qualitativ bei allen Rechnern das gleiche beschriebene Verhalten in den drei Phasen zu beobachten. Dies gilt ungeachtet der Tatsache, dass sich die einzelnen Rechner vor allem bezüglich der Hardware (Ausnahme *gruenau1–gruenau2*) stark unterscheiden. Detaillierte Angaben zu den einzelnen verwendeten Rechnern befinden sich in Anhang D.

6.4 Experimentserie: 1024 Philosophen mit Fibonaccizahlenberechnung

In einer weiteren Experimentserie wurde überprüft, ob die in der vorhergehenden Experimentserie beobachteten Abhängigkeiten der gemessenen Laufzeiten bzw. Time-Warps von der jeweiligen Belegung der Laufzeitparameter auch bei einer deutlichen Variation der Simulationsparameter auftreten.

Parameterbelegungen

Als erste Simulationsparameteränderung wurde die Anzahl der Philosophen deutlich erhöht und zwar auf den Wert 1024. Bei der Wahl dieser Parameterbelegung wurde erneut darauf geachtet, dass sich die Anzahl der Philosophen durch alle betrachteten Anzahlen von LPs teilen lässt.

Da sich durch die hohe Anzahl von Philosophenprozessen allerdings auch die jeweils benötigte Laufzeit der einzelnen Simulationsläufe stark erhöhte, musste der betrachtete Modellzeitraum eingeschränkt werden. Daher wurde in dieser Experimentserie die Simulationsendzeit bei 100 Zeiteinheiten festgelegt.

Die verwendeten Laufzeitparameterbelegungen waren hingegen die gleichen wie bei der Experimentserie mit 32 Philosophen.

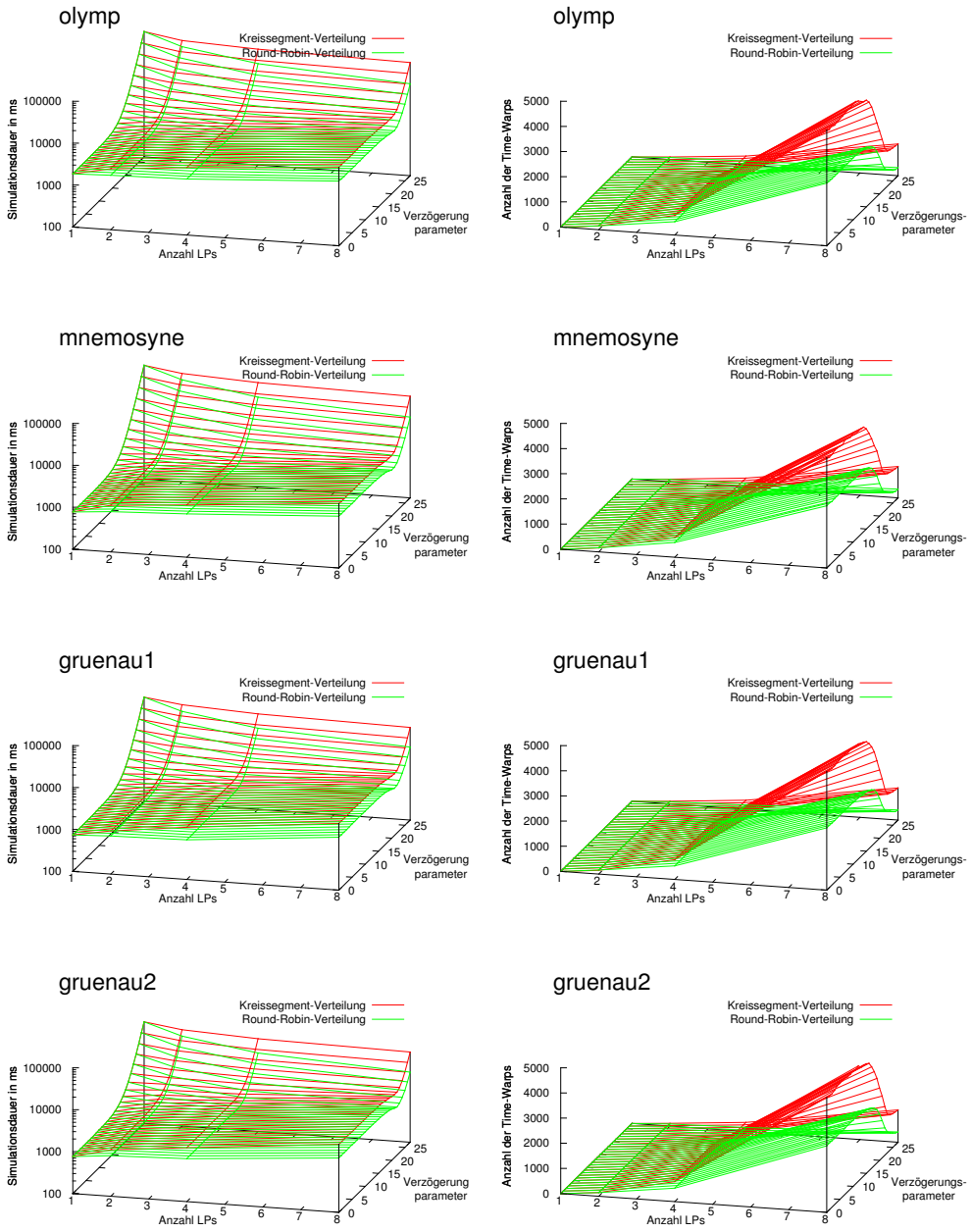

Abbildung 6.8: Vergleich der gemessenen Laufzeiten bzw. Time-Warps auf den Rechnern *olymp*, *mnemosyne*, *gruenau1* und *gruenau2* – 32 Philosophen, Fibonaccizahlenberechnung

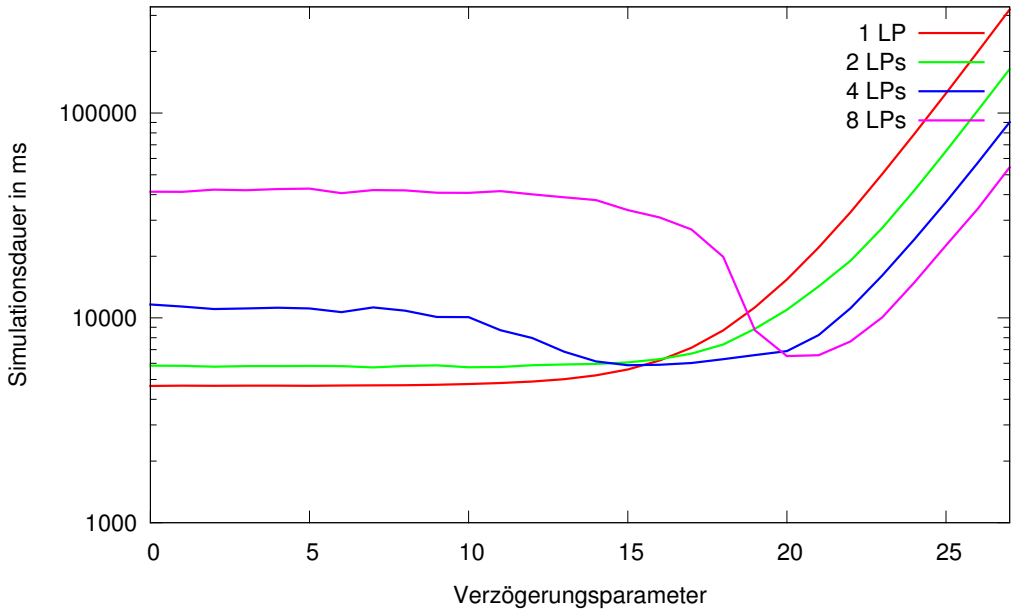

Abbildung 6.9: Gemessene Laufzeiten (1024 Philosophen, Fibonaccizahlenberechnung, Round-Robin-Verteilung)

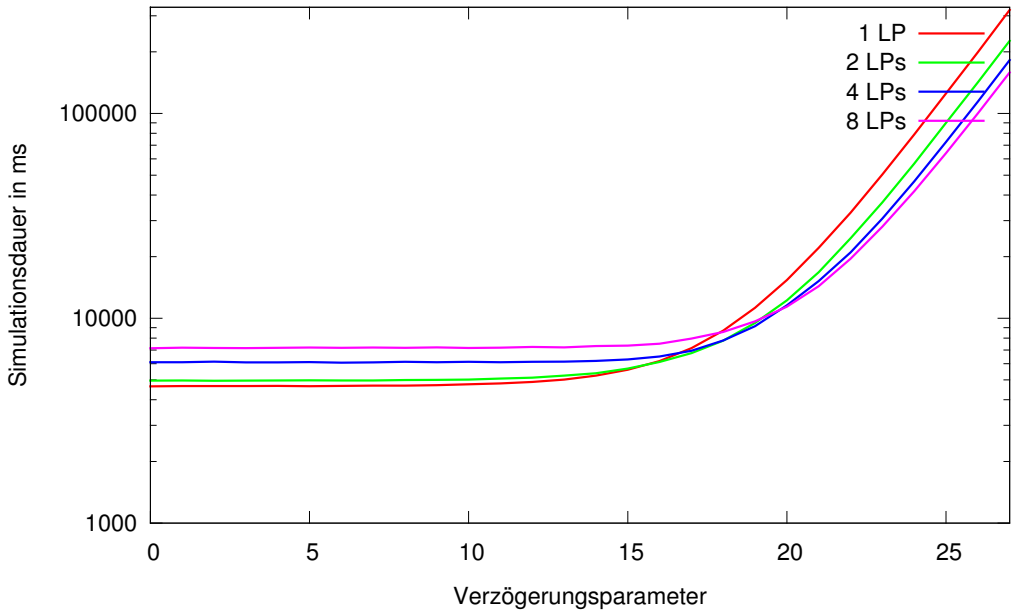

Abbildung 6.10: Gemessene Laufzeiten (1024 Philosophen, Fibonaccizahlenberechnung, Kreissegment-Verteilung)

Gemessene Laufzeiten und Time-Warps

Das Simulationsmodell wurde mit den veränderten Parameterbelegungen wiederholt auf *olymp* ausgeführt. Die gemessenen Laufzeiten sind in den Abbildungen 6.9 und 6.10 in Abhängigkeit von der Verzögerungsparameterbelegung dargestellt.

Obwohl sich die Diagramme auf den ersten Blick stark unterscheiden, ist das Verhalten bei beiden Verteilungen doch qualitativ das gleiche, wie das bei der Experimentserie mit 32 Philosophen für beide Verteilungen beschriebene. In beiden Diagrammen durchlaufen die Kurven die diskutierten drei Phasen. Sehr auffällig ist allerdings bei der Round-Robin-Verteilung die relativ hohe Laufzeit bei kleinen Verzögerungsparameterbelegungen und die daraus resultierende Größe der „Delle“ der Kurven für 4 und 8 LPs. War dieser zwischenzeitliche Einbruch der Laufzeiten bei den Experimenten mit 32 Philosophen nur sehr schwach ausgeprägt, findet hier zwischen den Verzögerungsparameterbelegungen 15 und 20, trotz Erhöhung des Rechenaufwandes eine Beschleunigung der Simulationsläufe etwa um den Faktor 2 statt.

Erklären lässt sich dieses Verhalten erneut mit einer Betrachtung der während dieser Experimente aufgetretenen Time-Warps. Die Anzahl der Time-Warps ist in den Abbildungen 6.11 und 6.12 jeweils in Abhängigkeit von der Verzögerungsparameterbelegung dargestellt. Dabei ist zu beachten, dass bei diesem Abbildungspaar ausnahmsweise unterschiedliche Skalierungen der Zeitachsen verwendet wurden.

Wie angesichts der auffällig hohen Laufzeiten bei der Round-Robin-Verteilung bei kleinen Verzögerungsparameterbelegungen zu vermuten war, ist auch die Anzahl der Time-Warps bei gleichen Laufzeitparameterbelegungen außerordentlich hoch. Im Verhältnis zu der Anzahl von Time-Warps bei der Kreissegment-Verteilung ist sie stellenweise sogar fast um den Faktor 40 höher.

Der Grund dafür liegt erneut darin, dass die Time-Warp-Wahrscheinlichkeit bei der Round-Robin-Verteilung bedingt durch die große Anzahl von Ressourcen, die von mehreren LPs geteilt werden, erheblich höher ist. Im Vergleich zu den Experimenten mit 32 Philosophen bearbeitet ein LP nun jedoch nicht nur jeweils 4 bis 16 Philosophenprozesse (bei 2–8 LPs), sondern 128 bis 512. Da die repräsentierten Philosophen jedoch weiterhin gleichmäßig um den Tisch verteilt sind, hat sich durch die größere Anzahl von Philosophenprozessen pro LP die Wahrscheinlichkeit, dass es an einer einzelnen geteilten Ressource einen Konflikt gibt, verringert. Infolgedessen tritt nun nicht mehr der bei 32 Philosophen zu beobachtende Effekt ein, dass alle LPs weitestgehend gleichmäßig in der Modellzeit voranschreiten, und dies führt letztendlich zu der höheren Anzahl von auftretenden Time-Warps.

Vergleich von Round-Robin- und Kreissegment-Verteilung

Die Auswirkungen der teilweise stark erhöhten Simulationslaufzeiten bei der Round-Robin-Verteilung werden besonders gut bei einem direkten Vergleich mit den gemessenen Zeiten aus Experimenten mit Kreissegment-Verteilung deutlich. In Abbildung 6.13 wurden die Simulationslaufzeiten der Experimente beider Verteilungen

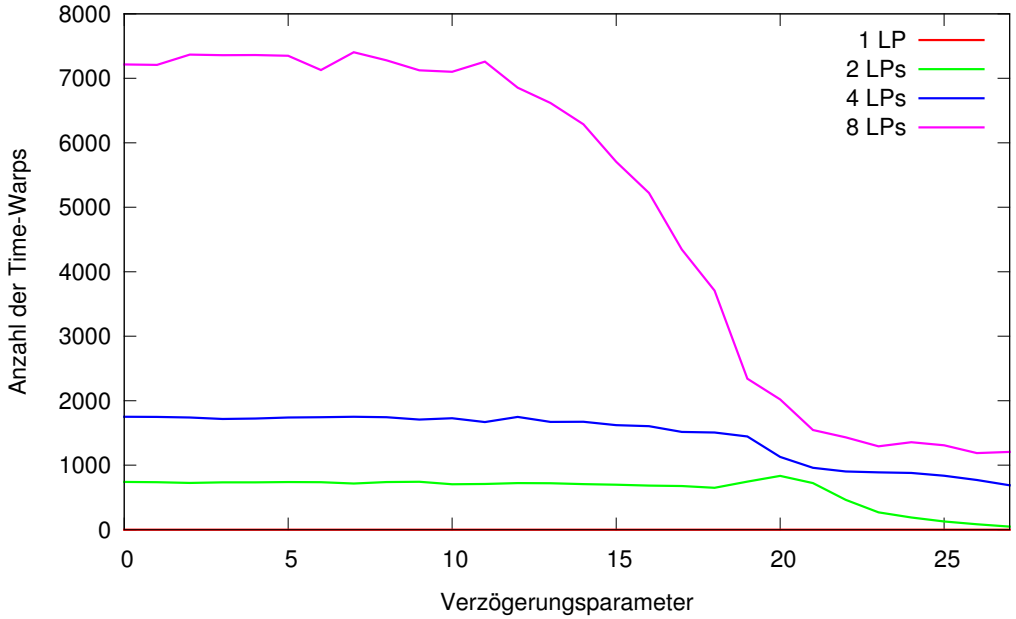

Abbildung 6.11: Gemessene Time-Warps (1024 Philosophen, Fibonaccizahlenberechnung, Round-Robin-Verteilung)

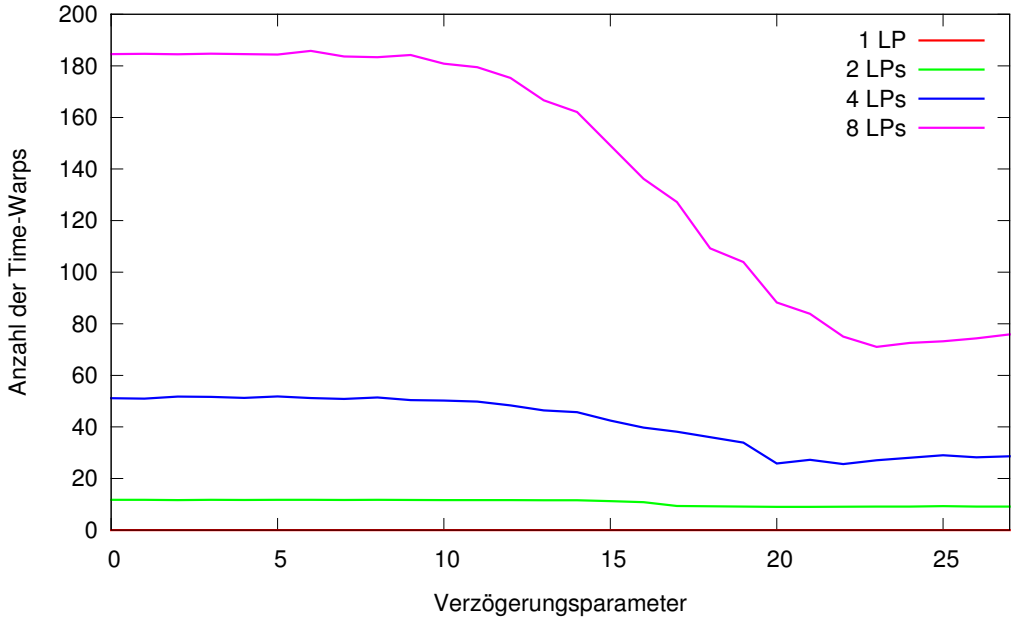

Abbildung 6.12: Gemessene Time-Warps (1024 Philosophen, Fibonaccizahlenberechnung, Kreissegment-Verteilung)

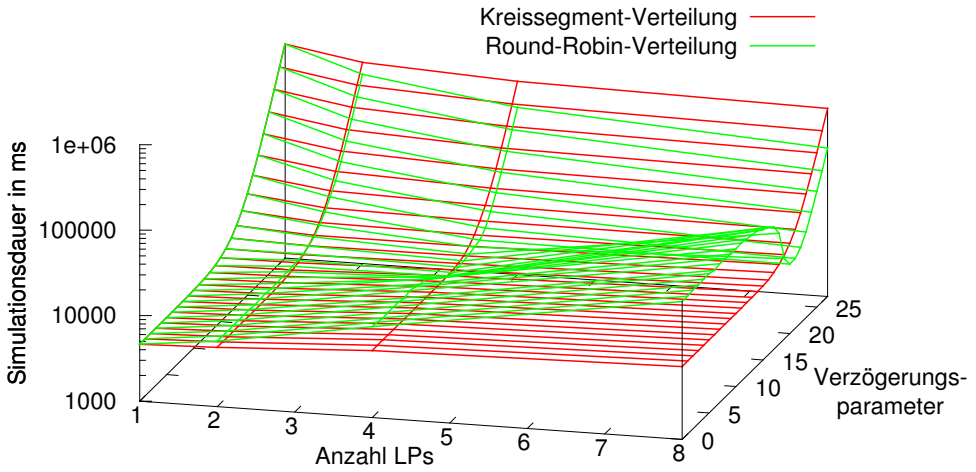

Abbildung 6.13: Gemessene Laufzeiten (1024 Philosophen, Fibonaccizahlenberechnung, Round-Robin- und Kreissegment-Verteilung)

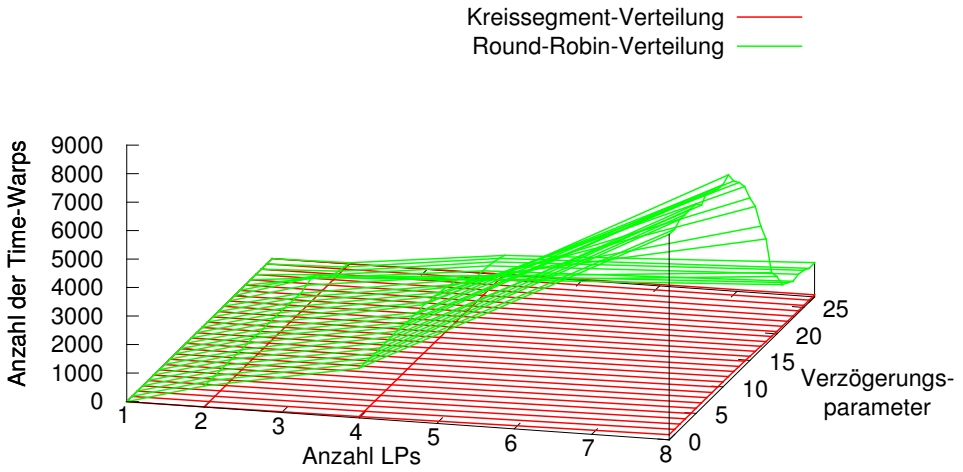

Abbildung 6.14: Anzahl der Time-Warps (1024 Philosophen, Fibonaccizahlenberechnung, Round-Robin- und Kreissegment-Verteilung)

zusammengefasst. Wie man gut erkennen kann, ist die Laufzeiterhöhung bei der Round-Robin-Verteilung so stark, dass für kleine und mittlere Verzögerungsparameterbelegungen die Simulationsausführung länger dauerte als die äquivalenten Experimente auf Basis der Kreissegment-Verteilung. Erst bei höheren Verzögerungsparameterbelegungen stellt die Round-Robin-Verteilung die effizientere Verteilung dar.

Interessanterweise gilt letztere Feststellung ungeachtet der Tatsache, dass auch bei höheren Verzögerungsparameterbelegungen bei der Round-Robin-Verteilung deutlich mehr Time-Warps auftreten als bei der Kreissegment-Verteilung. Dies ist vor allem in Abbildung 6.14, die die aufgetretenen Time-Warps der Experimente beider Verteilungen zusammenfasst, gut erkennbar.

Anhand dieser Tatsache wird deutlich, dass das Ausmaß der durch Time-Warps erzeugten Laufzeiterhöhungen nicht nur von deren Anzahl abhängt. Prinzipiell führt jeder Time-Warp eines LPs zu einer erneuten Berechnung der auf diesem LP seit der Rücksprungzeit fälschlicherweise durchgeführten Rechenoperationen. Daraus folgt, dass der zusätzliche Aufwand einer Time-Warp-Behandlung umso größer ist, je weiter ein zurücksetzender LP gegenüber dem Time-Warp veranlassenden LP in der Modellzeit vorangeschritten ist. In dem hier betrachteten Fall führt dies dazu, dass bei gleichen, hohen Verzögerungsparameterbelegungen trotz höherer Anzahl auftretender Time-Warps die Simulationslaufzeiten der Round-Robin-Verteilung geringer sind als bei der Kreissegment-Verteilung.

Vergleich der Ergebnisse von verschiedenen Rechnern

Analog zur Experimentserie mit 32 Philosophen wurde auch diese Experimentserie auf drei weiteren Rechnern wiederholt, um die Unabhängigkeit der beschriebenen Phänomene von Hardware und Betriebssystem zu überprüfen. In Abbildung 6.15 sind untereinander die gemessenen Simulationslaufzeiten bzw. Time-Warps der durchgeführten Experimente auf den Rechnern *olymp*, *mnemosyne*, *gruenau1* und *gruenau2* zu sehen. Wie zu erwarten, ist das beobachtete Verhalten bei allen Rechnern das gleiche.

6.5 Experimentserie: 32 Philosophen mit MD5-Summenberechnung

Ein weiterer zu untersuchender Punkt bestand in der Fragestellung, wie sich die benötigten Laufzeiten von MYTIMEWARP-Simulationsimplementationen im Vergleich zu denen von Implementationen auf Basis einer existierenden, sequentiellen Simulationsbibliothek verhalten. Als Vergleichsobjekt wurde dafür die Simulationsbibliothek DESMO-J gewählt, die in Anhang B.1 etwas detaillierter vorgestellt wird.

Allerdings musste in Vorexperimenten festgestellt werden, dass die bisher gewählte künstliche Verzögerung durch Fibonaccizahlenberechnung für den beabsichtigten Vergleich ungeeignet ist. Wie in Anhang C.2 etwas ausführlicher dargestellt, wird die

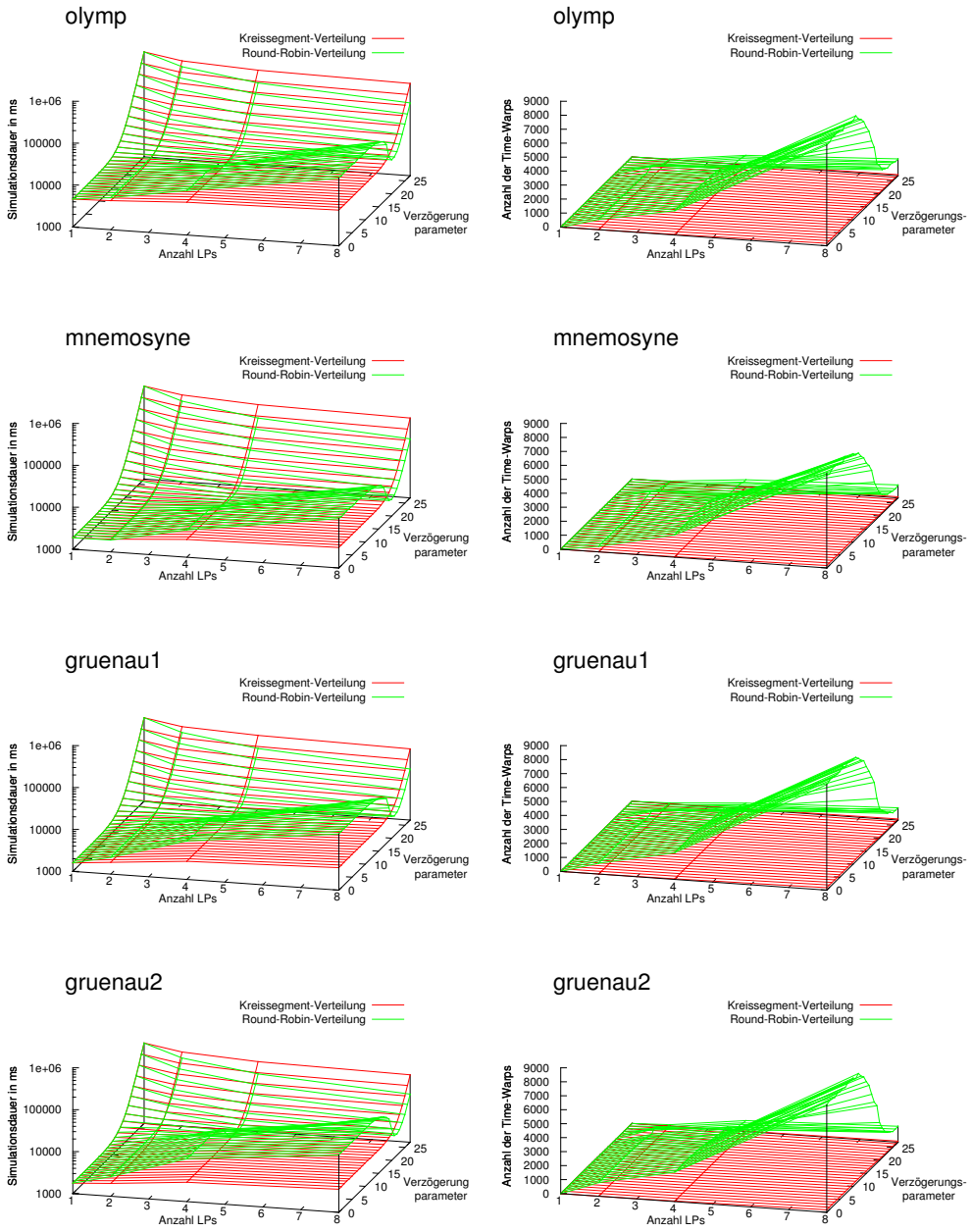

Abbildung 6.15: Vergleich der gemessenen Laufzeiten bzw. Time-Warps auf den Rechnern *olymp*, *mnemosyne*, *gruenau1* und *gruenau2* – 1024 Philosophen, Fibonaccizahlenberechnung

rekursive Berechnung der Fibonaccizahlen je nach verwendeter Simulationsbibliothek unterschiedlich stark durch die JVM optimiert. Dadurch ist jedoch nicht mehr gewährleistet, dass mit gleichen Verzögerungsparameterbelegungen auch stets der gleiche zusätzliche Rechenaufwand verbunden ist, was die Vergleichbarkeit der Ergebnisse von Implementationen auf Basis unterschiedlicher Simulationsbibliotheken stark einschränkt. Als Ausweg wurde für den angestrebten Vergleich der Simulationsbibliotheken auf die künstliche Verzögerung durch MD5-Summenberechnung zurückgegriffen, die von dieser Problematik nicht betroffen ist.

Parameterbelegungen

Die Parameterbelegungen waren bis auf eine Ausnahme bei allen durchgeführten Simulationsläufen die gleichen wie bei der ersten Experimentserie der Philosophen mit Fibonaccizahlenberechnung: Der betrachtete Modellzeitraum erstreckte sich von 0 bis 1000 Zeiteinheiten, die Anzahl der Philosophen war stets 32 und es wurden bei der MYTIMEWARP-Experimentserie nacheinander Simulationsläufe unter Verwendung von 1, 2, 4 und 8 LPs durchgeführt.

Lediglich die Belegung des Verzögerungsparameters wurde aufgrund des veränderten Verzögerungsverfahrens deutlich anders gewählt und variiert. Der betrachtete Verzögerungsparameterraum erstreckte sich von 0 bis 5000. Dabei erfolgte die Variation nicht in äquidistanten, sondern stufenweise wachsenden Intervallen: von 0 bis 100 in 10er-Schritten, von 100 bis 500 in 50er-Schritten und 500 bis 5000 in 500er-Schritten.

Ergebnisse der MYTIMEWARP-Implementation

Die erste Serie von Simulationsläufen wurde mit der MYTIMEWARP-Implementation auf *olymp* unter Verwendung beider Verteilungen durchgeführt. Die dabei gemessenen Laufzeiten sind in den Abbildungen 6.16 und 6.17 dargestellt.

Der erste offensichtliche Unterschied zu den vorherigen Experimenten mit der künstlichen Verzögerung durch Fibonaccizahlenberechnung besteht darin, dass der Aufwand für die wiederholte MD5-Summenberechnung langfristig nicht mehr exponentiell, sondern nur linear mit der Verzögerungsparameterbelegung wächst.

Analog zu den vorherigen Experimentserien ist hingegen gut zu beobachten, dass bei kleinen Verzögerungsparameterbelegungen zusätzliche LPs zu einer Zunahme der benötigten Simulationslaufzeit führen, während diese bei hohen Verzögerungsparameterbelegungen zu einer Beschleunigung der Simulation führen.

Dass die Ursache für dieses Phänomen auch hier in der Anzahl der aufgetretenen Time-Warps liegt, lässt sich anhand der gemessenen Werte bestätigen, die in den Diagrammen in den Abbildungen 6.18 und 6.19 dargestellt sind. Die bereits bei der Diskussion der vorherigen Experimentserien erfolgten Begründungen für dieses Verhalten gelten weiterhin.

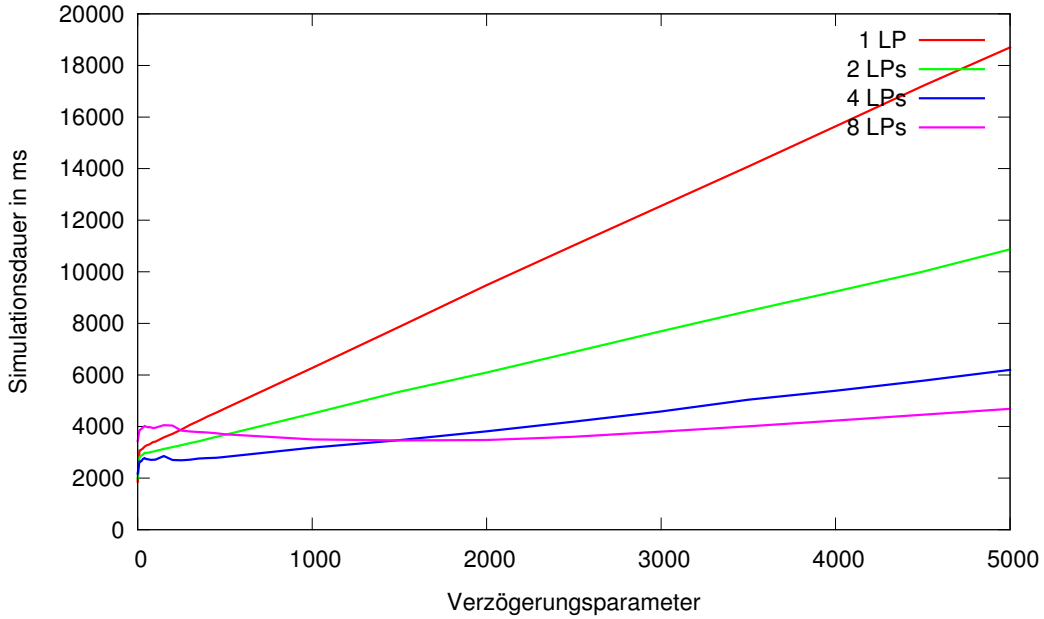

Abbildung 6.16: Gemessene Laufzeiten (32 Philosophen, MD5-Summenberechnung, Round-Robin-Verteilung)

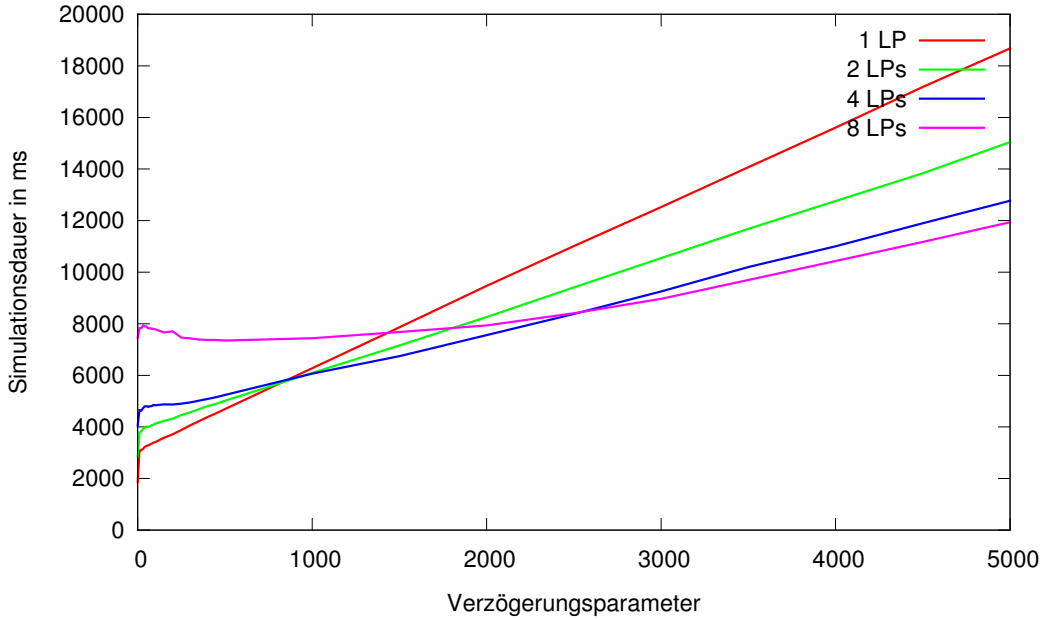

Abbildung 6.17: Gemessene Laufzeiten (32 Philosophen, MD5-Summenberechnung, Kreissegment-Verteilung)

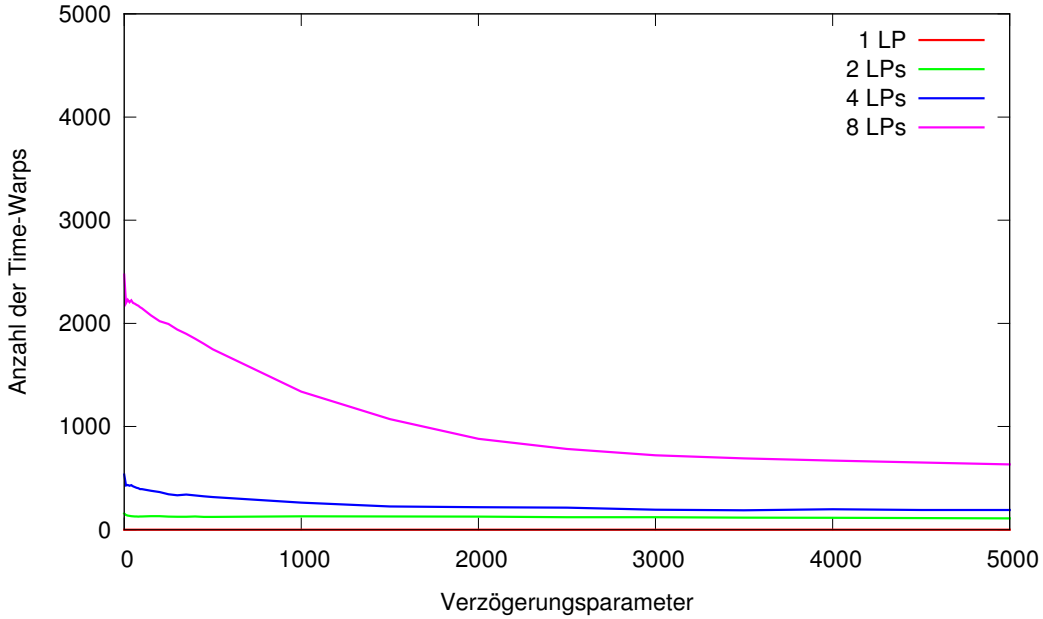

Abbildung 6.18: Gemessene Time-Warps (32 Philosophen, MD5-Summenberechnung, Round-Robin-Verteilung)

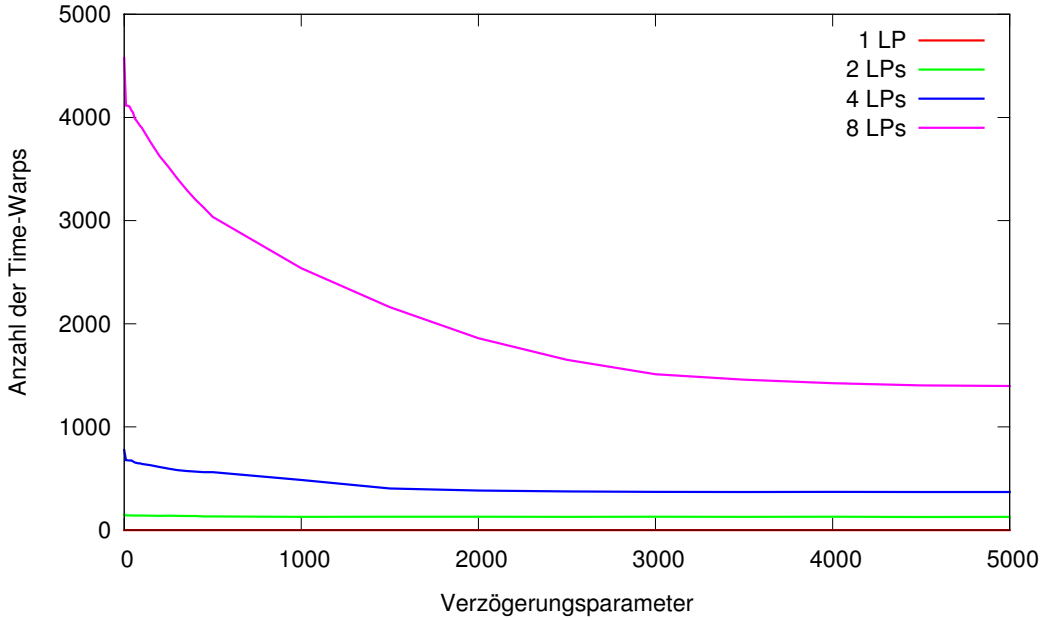

Abbildung 6.19: Gemessene Time-Warps (32 Philosophen, MD5-Summenberechnung, Kreissegment-Verteilung)

Ergebnisse der DESMO-J-Implementation

Um den beabsichtigten Vergleich der beiden Simulationsbibliotheken durchzuführen, wurde das beschriebene Simulationsmodell auf Basis der Simulationsbibliothek DESMO-J erneut implementiert, wobei die MYTIMEWARP-Simulationsimplementation als Ausgangspunkt genommen wurde. Dabei war der Implementationsaufwand aufgrund der sehr ähnlichen Schnittstellen der Simulationskerne beider Simulationsbibliotheken außerordentlich gering.

Anschließend wurde die Experimentserie der 32 dinierenden Philosophen mit MD5-Summenberechnung erneut mit der DESMO-J-Simulationsimplementation durchgeführt. Aufgrund der Tatsache, dass der Simulationskern von DESMO-J sequentiell arbeitet, entfielen dabei die Parameter *numberOfLPs* und *roundRobin*. Die verwendeten Belegungen der restlichen Parameter waren identisch mit denen der Experimente, die auf Basis der MYTIMEWARP-Simulationsimplementation durchgeführt wurden.

Die bei den Experimenten ermittelten Simulationslaufzeiten sind in den Diagrammen in den Abbildungen 6.20 und 6.21 in Abhängigkeit von der Verzögerungsparameterbelegung dargestellt (jeweils die hervorgehobene Kennlinie). Wie zu erwarten, steigt auch bei der Verwendung von DESMO-J die benötigte Simulationslaufzeit langfristig linear mit der Belegung des Verzögerungsparameters.

Vergleich der Ergebnisse beider Simulationsimplementationen

Neben den Simulationslaufzeiten der DESMO-J-Implementation wurden in den Abbildungen 6.20 und 6.21 auch die gemessenen Laufzeiten der MYTIMEWARP-Simulationsimplementation unter Verwendung von 1, 2, 4 und 8 LPs eingezeichnet. Dabei stellen diese zusätzlichen Kennlinien in Abbildung 6.20 die Simulationsergebnisse der Experimente mit Round-Robin-Verteilung dar, während die Ergebnisse der Experimente mit Kreissegment-Verteilung in Abbildung 6.21 zu sehen sind.

Vergleicht man zunächst die Ergebnisse der DESMO-J-Experimente mit denen der MYTIMEWARP-Experimente unter Verwendung eines einzelnen LPs, so stellt man fest, dass die MYTIMEWARP-Simulationsimplementation stets langsamer ist als die DESMO-J-Implementation. Dies ist dadurch begründet, dass auch ohne das Auftreten von Time-Warps, schon allein durch die ständige Vorbereitung auf das potentielle Eintreten eines solchen, die MYTIMEWARP-Implementation zusätzlich belastet ist. Vor allem die ständige Sicherung des jeweils aktuellen Modellzustandes spielt hier als Zusatzaufwand für die MYTIMEWARP-Implementation eine wesentliche Rolle.

Diese Situation ändert sich, sobald die MYTIMEWARP-Simulation auf mehrere LPs zurückgreifen kann. Wie in beiden Diagrammen zu sehen ist, benötigt die MYTIMEWARP-Simulationsimplementation durch die parallele Berechnung der MD5-Summen bei größerer Rechenlast stets weniger Ausführungszeit als die DESMO-J-Implementation. Dabei gilt, dass die Simulation umso schneller durchgeführt werden kann, je mehr LPs verwendet werden.

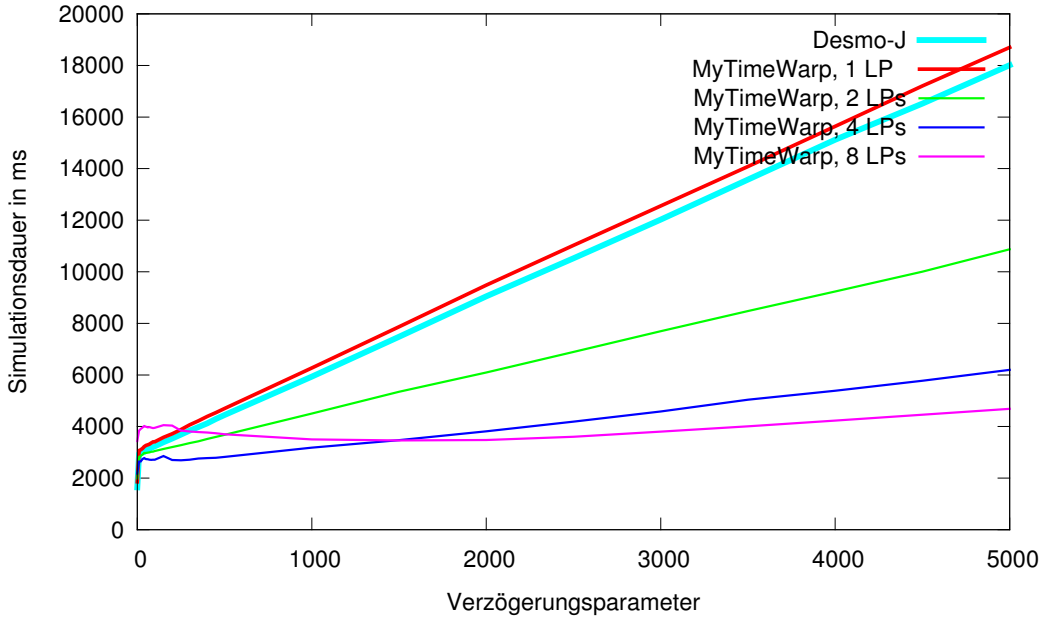

Abbildung 6.20: Vergleich von DESMO-J und MYTIMEWARP (32 Philosophen, MD5-Summenberechnung, Round-Robin-Verteilung)

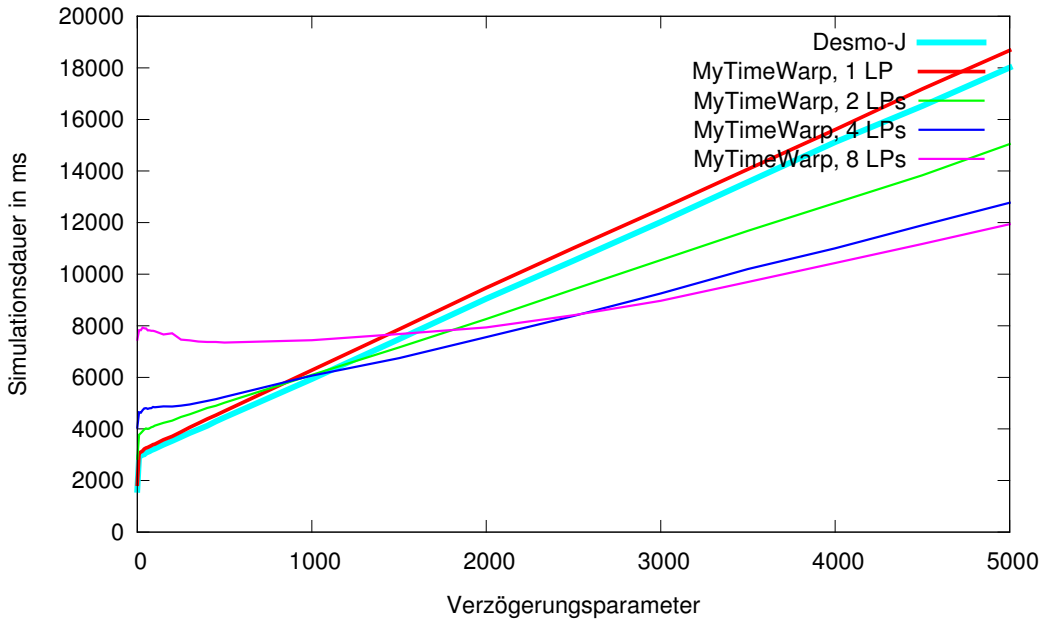

Abbildung 6.21: Vergleich von DESMO-J und MYTIMEWARP (32 Philosophen, MD5-Summenberechnung, Kreissegment-Verteilung)

Problematisch ist hingegen die Situation bei kleinen Verzögerungsparameterbelegungen. Wie schon in den Auswertungen der Experimentserien auf Basis der Fibonaccizahlenberechnung ausgeführt wurde, steigt hier bei der MYTIMEWARP-Implementation die benötigte Simulationslaufzeit durch die zahlreicher auftretenden Time-Warps an, ohne dass dies durch die Beschleunigung der MD5-Summenberechnung kompensiert werden kann. Infolgedessen benötigt die DESMO-J-Implementation hier im Vergleich deutlich weniger Zeit.

Vergleich der Ergebnisse von verschiedenen Rechnern

Analog zu den Experimentserien mit Fibonaccizahlenberechnung wurden auch die Experimentserien mit MD5-Summenberechnung auf drei weiteren Rechnern wiederholt, um die Unabhängigkeit der beschriebenen Phänomene von Hardware und Betriebssystem zu überprüfen.

In Abbildung 6.22 sind untereinander die gemessenen Simulationslaufzeiten der durchgeführten Experimente auf den Rechnern *olymp*, *mnemosyne*, *gruenau1* und *gruenau2* unter Verwendung der Round-Robin- (links) und der Kreissegment-Verteilung (rechts) zu sehen. Erwartungsgemäß sind die Ergebnisse auf allen Rechnern qualitativ die gleichen.

6.6 Zusammenfassung der Experimentergebnisse

Als wichtigstes Ergebnis der durchgeführten Experimentserien ist festzuhalten, dass die optimistisch-parallele Simulationsausführung tatsächlich zu einer wirksamen Beschleunigung von Simulationsläufen führen kann. Allerdings ist die bloße Verteilung des Simulationsmodells auf mehrere Recheneinheiten allein noch kein Garant für eine Verkürzung der Simulationslaufzeit. Wie in den Experimenten ebenfalls bestätigt werden konnte, kann eine Parallelisierung die Simulationslaufzeit auch deutlich erhöhen.

Im Wesentlichen sind bei einer parallelen Ausführung von Simulationsläufen der prinzipielle Erfolg und das Ausmaß der Simulationsbeschleunigung von drei Faktoren abhängig:

- wie aufwendig die Berechnung der einzelnen Simulationsschritte ist,
- wie lose gekoppelt (räumlich und/oder zeitlich) die einzelnen Bestandteile des Simulationsmodells sind und
- wie gut die lose bzw. enge Kopplung zwischen Modellbestandteilen identifiziert und bei der Verteilung der Teile auf Recheneinheiten berücksichtigt wird bzw. werden kann.

Alle drei genannten Punkte sind stark vom jeweiligen Simulationsmodell abhängig – die ersten beiden direkt, der dritte indirekt. Insofern ist die erste Frage, die bei einer beabsichtigten Parallelisierung einer Simulation zu beantworten ist, ob sich das Simulationsmodell gut oder zumindest überhaupt für eine parallelisierte

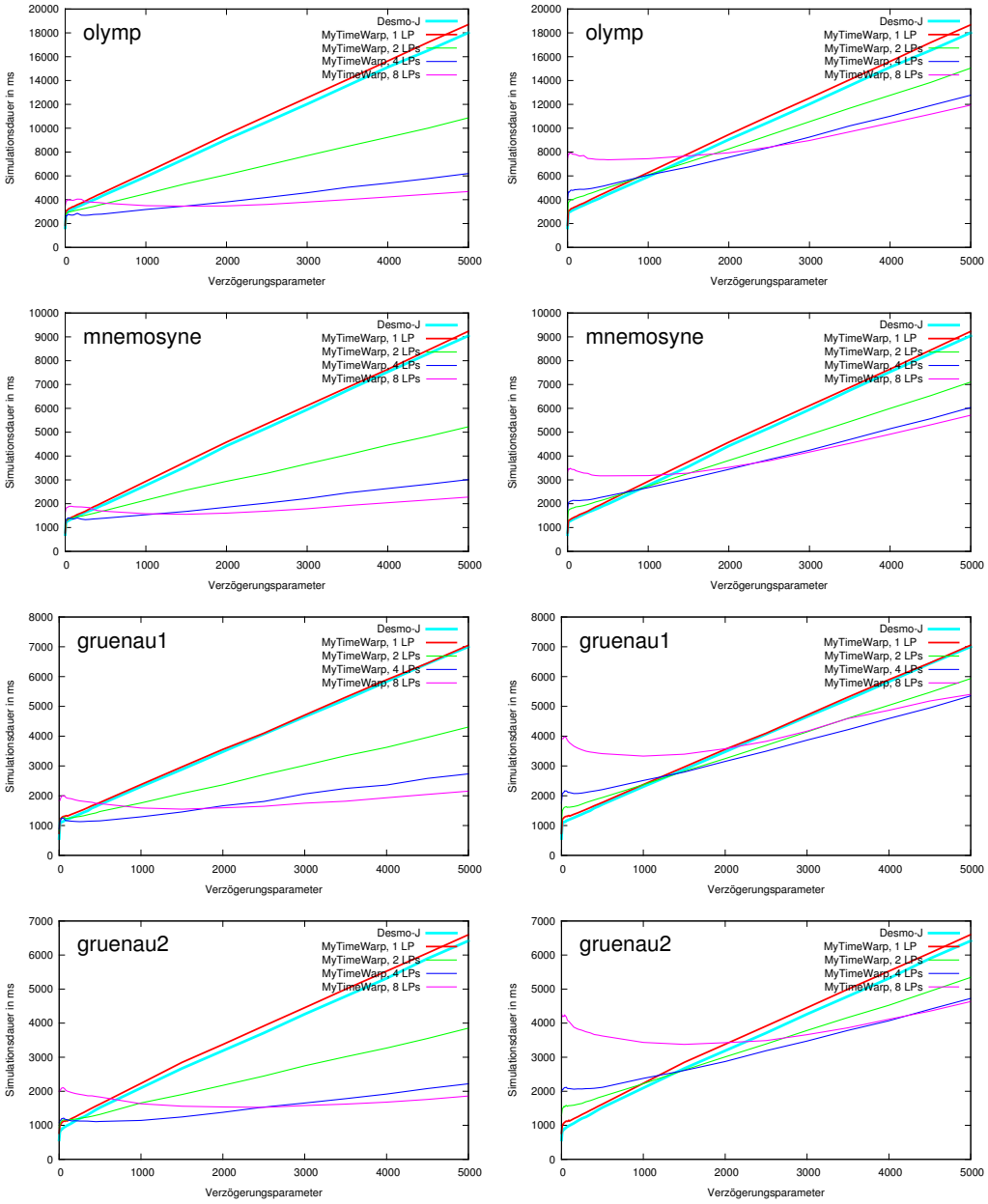

Abbildung 6.22: Vergleich der gemessenen Laufzeiten auf den Rechnern *olymp*, *mnemosyne*, *gruenau1* und *gruenau2* jeweils unter Verwendung der Round-Robin- (links) und der Kreissegment-Verteilung (rechts) – 32 Philosophen, MD5-Summenberechnung

Ausführung eignet. Für das unmodifizierte Simulationsmodell der dinierenden Philosophen lässt sich diese Frage eindeutig negativ beantworten. Ohne die künstliche Verzögerung ist bei allen durchgeführten Experimenten die sequentielle Simulationsausführung am schnellsten gewesen.

Der dritte genannte Punkt weist jedoch bereits darauf hin, dass selbst das Vorliegen eines gut parallelisierbaren Simulationsmodells allein noch keine hohe Simulationsbeschleunigung garantiert. Eine schlechte Verteilung der Modellbestandteile auf die verfügbaren Recheneinheiten kann immer noch zu einer geringen oder gar negativen Beschleunigung der Simulationsausführung führen. Dies konnte auch in den durchgeführten Experimentserien nachgewiesen werden und zwar beim jeweiligen Vergleich der gemessenen Ergebnisse von Experimenten auf Basis der Round-Robin-Verteilung mit denen der Experimente mit Kreissegment-Verteilung.

Ein weiteres Ergebnis der Experimente zu diesem Punkt war allerdings auch, dass das Finden einer guten Verteilung, selbst bei einem sehr einfachen Simulationsmodell wie dem hier verwendeten, mitnichten trivial ist. Exemplarisch sei hier die Experimentserie der 1024 Philosophen mit Fibonaccizahlenberechnung genannt, bei dem bei der Verwendung von 8 LPs für kleine Verzögerungsparameterbelegungen die Kreissegment-Verteilung die günstigere darstellt, während bei größerem Rechenaufwand die Round-Robin-Verteilung zu schnelleren Simulationsläufen führt.

Des Weiteren konnte die Leistungsfähigkeit der prototypischen Simulationsbibliothek MYTIMEWARP im Vergleich mit der sequentiellen Simulationsbibliothek DESMO-J überprüft werden. Dabei muss betont werden, dass die Aussagekraft dieses Vergleichs begrenzt ist. In der aktuellen, prototypischen Implementation von MYTIMEWARP fehlen noch zahlreiche Funktionalitäten, deren Nachrüstung sich definitiv negativ in der Simulationslaufzeit mit MYTIMEWARP umgesetzter Simulationsimplementationen auswirken werden. Demgegenüber stellt DESMO-J eine vollständige, für den Produktionsbetrieb geeignete Simulationsbibliothek dar.

Trotz dieser Einschränkung sind die Ergebnisse zufriedenstellend. Wie aufgrund des Mehraufwands bei der optimistisch-parallelen Simulationsausführung zu erwarten war, benötigen die Simulationsimplementationen von MYTIMEWARP bei sequentieller Ausführung (Verwendung eines einzelnen LPs) etwas mehr Simulationslaufzeit als äquivalente Implementationen von DESMO-J. Bei der Verwendung mehrerer LPs stellte sich weitestgehend der Effekt ein, dass eine MYTIMEWARP-Simulationsimplementation genau dann schneller war als eine Implementation mit DESMO-J, wenn die parallele Ausführung der MYTIMEWARP-Implementation schneller war als eine sequentielle Ausführung. Wie bereits beschrieben, war dies vor allem bei hohen Verzögerungsparameterbelegungen der Fall.

Neben der Beschleunigung der Simulationsausführung war das zweite erklärte Ziel bei der Entwicklung von MYTIMEWARP die Unterstützung prozessorientierter Simulationsmodelle mit möglichst wenigen Einschränkungen für den Simulationsmodellierer. Wie sowohl bei einer näheren Betrachtung des Simulationsmodells der dinierenden Philosophen als auch der weiteren, in Anhang A.3 vollständig abgedruckten Quelltexte gut erkennbar ist, wurde auch dieses Ziel erreicht.

Die Nähe der von MYTIMEWARP verlangten Simulationsmodelle zu sequentiellen Simulationsmodellen konnte bei den durchgeführten Experimenten auch bereits erfolgreich ausgenutzt werden und zwar bei der Erstellung der DESMO-J-Implementation. Diese wurde auf der Basis der MYTIMEWARP-Implementation erstellt, wobei die wesentlichen Änderungen im lediglichen Austausch der Namen der aufgerufenen Methoden des Simulationskerns bestanden.

Zusammengefasst lässt sich also festhalten, dass die bisherigen Erfahrungen mit der prototypischen Implementation von MYTIMEWARP zufriedenstellend sind. Insofern erscheint eine Weiterentwicklung und insbesondere Komplettierung der Simulationsbibliothek sinnvoll.

KAPITEL 7

ZUSAMMENFASSUNG

Das zentrale Ziel dieser Arbeit bestand in der Konzeption einer Simulationsbibliothek, deren Simulationskern die optimistisch-parallele Ausführung prozessorientiert erstellter Simulationsmodelle unterstützt. Dadurch sollten die Vorteile von paralleler Simulation und prozessorientierter Modellierung, eine beschleunigte Simulationsausführung bei gleichzeitig intuitiven und zum Ausgangsszenario strukturell äquivalenten Simulationsmodellen, kombiniert werden.

Zu diesem Zweck wurden im Rahmen dieser Arbeit die prinzipiellen Herausforderungen einer entsprechenden Umsetzung analysiert und auch die sich erst bei einer konkreten Implementation ergebenden Probleme untersucht. Anschließend wurde eine prototypische Umsetzung namens MyTimeWarp in der Programmiersprache Java realisiert. Auf Basis von MyTimeWarp wurden verschiedene Simulationsexperimente durchgeführt, um zu überprüfen, inwieweit die avisierten Ziele erreicht wurden.

7.1 Implementation optimistisch-paralleler Prozesse

Anforderungen an Coroutinen/Continuation-Implementationen

Die Grundlage für jede Unterstützung prozessorientierter Simulationsmodelle durch einen Simulationskern ist eine generische Prozessimplementation. Nur auf deren Basis können in einem Simulationsmodell konkrete Prozesse verwendet werden, deren Verhalten der Simulationsmodellierer uneingeschränkt definieren kann.

Dabei ist jede Umsetzung einer generischen Prozessimplementation eng mit den beiden programmiersprachlichen Konzepten der Coroutinen und Continuations verbunden. Dies führte in der vorliegenden Arbeit zu einer Nebenproblematik, da Coroutinen/Continuations in den Sprachstandards vieler aktueller Programmiersprachen, so auch der avisierten Implementationssprache Java, nicht vorhanden sind.

Allerdings existieren zahlreiche generische Implementationsansätze für Coroutinen/Continuations, die sich in vielen Fällen auch in diesen Programmiersprachen umsetzen lassen.

Um einen geeigneten Implementationsansatz zu finden, wurden in dieser Arbeit zunächst generische Coroutinen- und Continuation-Implementationen untersucht. Dabei wurden diese anhand charakteristischer Eigenschaften klassifiziert und anschließend auf Basis dieser Klassifikation diejenigen Anforderungen identifiziert, die bei der Verwendung für eine Implementation optimistisch-paralleler Prozesse zwingend benötigt werden. Dabei wurden die Implementationen von Coroutinen und Continuations als Einheit betrachtet – jede Implementation des einen Konzeptes ist bzw. enthält automatisch eine Implementation des anderen Konzeptes.

Die Zusammenfassung der identifizierten Minimalanforderungen stellt ein wesentliches Zwischenergebnis dieser Arbeit dar. Dabei ist zu betonen, dass die ermittelten Anforderungen unabhängig von einer konkreten Implementationssprache erfüllt sein müssen. Eine Coroutinen/Continuation-Implementation für die Umsetzung optimistisch-paralleler Prozesse muss folgende Eigenschaften aufweisen:

- Asymmetrie: die implementierten Coroutinen müssen sich bezüglich der aufrufenden Routine wie eine normale Subroutine verhalten,
- Unbeschränktheit: die Coroutinen dürfen bezüglich ihrer Verwendung keinen Einschränkungen unterworfen sein,
- Stackerhaltung: Aufrufe zur Unterbrechung einer Coroutine müssen nicht nur innerhalb selbiger, sondern auch in von der Coroutine aufgerufenen Subroutinen erfolgen dürfen,
- Lokalität: die zur Laufzeit generierten Continuations sollen mitnichten den Laufzeitzustand des gesamten Programmes, sondern nur den aktuellen Zustand der jeweiligen Coroutine umfassen und
- Mehrfachfortsetzbarkeit: eine einmal gespeicherte Continuation muss beliebig oft fortgesetzt werden können.

Coroutinen/Continuation-Implementation in Java

Basierend auf den ermittelten Anforderungen, wurden konkrete Implementationsansätze untersucht und deren Eignung bezüglich der Realisierung optimistisch-paralleler Prozesse analysiert. Während dabei für andere Programmiersprachen mehrere geeignete Implementationsansätze gefunden werden konnten, stellte sich die vorherige Wahl der Sprache Java als Zielsprache für die Implementation im Rahmen dieser Arbeit als problematisch heraus. Es existierte zwar bereits ein verbreiteter Implementationsansatz in Java, der auch schon in mehreren sequentiellen Simulationskernen auf Java-Basis (DESMO-J, JDISCO, JAVASIMULATION) erfolgreich verwendet wurde. Dieser Ansatz ist jedoch aufgrund der fehlenden Eigenschaft der Mehrfachfortsetzbarkeit bezüglich der vorher identifizierten Minimalanforderungen für optimistisch-parallele Prozesse ungeeignet.

Schließlich konnte im simulationsfernen Gebiet der Webapplikationsentwicklung nicht nur ein geeigneter Implementationsansatz, sondern sogar eine existierende und voll funktionsfähige Umsetzung gefunden werden. Dieser Ansatz basiert auf der nachträglichen Modifikation des vom Java-Compiler generierten Bytecodes. Dank der existierenden Realisierung dieses Ansatzes im Webapplikationsframework JAVAFLOW konnte eine vergleichsweise unkomplizierte generische Prozessimplementation im Simulationskern von MYTIMEWARP realisiert werden.

7.2 Die Simulationsbibliothek MYTIMEWARP

Bewertung der Anforderungen an Simulationsmodelle

Das erste Ziel bei der Entwicklung von MYTIMEWARP war die Unterstützung prozessorientierter Simulationsmodelle. Mit dem Hintergrund, die parallele Arbeitsweise des Simulationskerns weitestgehend vor dem Simulationsmodellierer zu verbergen, sollten diese Modelle eine größtmögliche Ähnlichkeit zu sequentiellen Simulationsmodellen aufweisen. Zu diesem Zwecke wurden bei der Definition der Schnittstellen des Simulationskerns bewusst die Schnittstellen sequentieller Simulationskerne (DESMO-J, ODEM, ODEMX) als Orientierung herangezogen.

Wie man unter anderem an den Modellbeispielen in Anhang A gut erkennen kann, unterscheiden sich die Simulationsmodelle von MYTIMEWARP nur unwesentlich von sequentiellen Simulationsmodellen. Des Weiteren zeichnet sich MYTIMEWARP bezüglich der Unterstützung intuitiver Simulationsmodelle, vor allem gegenüber existierenden parallelen Simulationsbibliotheken, durch folgende Eigenschaften aus:

- Im Simulationsmodell werden, analog zu sequentiellen Modellen, ausschließlich skalare Modellzeiten verwendet. Die innerhalb des Simulationskerns verwendete, strukturierte Modellzeit wird vor dem Simulationsmodellierer vollständig verborgen.
- Die interne Arbeitsweise des Simulationskerns, die auf LPs und dem Austausch von Nachrichten und Anti-Nachrichten zwischen selbigen basiert, ist ebenfalls für den Simulationsmodellierer unsichtbar.

  Stattdessen imitiert der Simulationskern in seinen Schnittstellen das Verhalten eines sequentiellen Simulationskerns mit einem zentralen Ereigniskalender und einer darüber iterierenden Abarbeitungsschleife vor. Der Simulationsmodellierer muss lediglich die Anzahl der LPs angeben und welche Prozesse/ Ereignisse auf welchem LP ausgeführt werden sollen. Beide Festlegungen beeinflussen jedoch nur die Ausführungsgeschwindigkeit – nicht das Simulationsergebnis.
- Analog bleibt dem Benutzer das Auftreten von Time-Warps sowie deren Behandlung vollständig verborgen. Dank der serialisierten Ausgabe erscheinen die Simulationsergebnisse so, als wären sie das Produkt eines sequentiellen Simulationslaufes.

- Bei der Erstellung von Simulationsmodellen können die beiden üblichen Modellierungskonzepte Ereignis und Prozess verwendet werden. LPs treten als Modellierungselement gar nicht auf, sondern dienen lediglich der Steuerung der Ausführung.
- Das Verbot der gleichzeitigen Referenzierung von Variablen durch mehrere LPs konnte zwar nicht aufgehoben werden. Seine Auswirkungen werden jedoch durch die Bereitstellung eines generischen Ressourcenkonzeptes deutlich abgemildert.

Basierend auf diesen Eigenschaften kann das erste avisierte Ziel, die Unterstützung intuitiver, prozessorientierter Simulationsmodelle, als weitestgehend erreicht bezeichnet werden.

Bewertung der erreichten Ausführungsgeschwindigkeit

Das zweite Ziel der Entwicklung von MYTIMEWARP lag in der Beschleunigung von Simulationsläufen durch eine optimistisch-parallele Ausführung des Simulationsmodells. Auch wenn die vorliegende Implementation ihren prototypischen Charakter weder verbergen kann noch soll, ließen sich mit ihrer Hilfe bereits erste Untersuchungen bezüglich dieses Ziels durchführen. Als Grundlage dieser Untersuchungen wurden verschiedene Experimente mit einem Simulationsmodell durchgeführt, dessen Rechenaufwand durch die Belegung eines Laufzeitparameters konfigurierbar war.

Aufgrund der gemessenen Simulationslaufzeiten lässt sich bestätigen, dass sowohl die optimistisch-parallele Simulation im Allgemeinen als auch deren konkrete Implementation im Simulationskern von MYTIMEWARP durchaus in der Lage sind, Simulationsausführungen stark zu beschleunigen. Allerdings kann der Erfolg einer Simulationsparallelisierung nicht von vornherein für alle Simulationsszenarien garantiert werden.

Wie bei der Zusammenfassung der Experimentergebnisse in Abschnitt 6.6 detailliert beschrieben wurde, hängt der Grad der Beschleunigung im Wesentlichen von drei Faktoren ab:

- dem Aufwand bei der Berechnung der einzelnen Simulationsschritte,
- dem Grad der Kopplung innerhalb des Simulationsmodells und
- der Verteilung der einzelnen Modellbestandteile auf die verfügbaren Recheneinheiten.

Bezüglich der ersten beiden Punkte lässt sich festhalten, dass ein Simulationsmodell grundsätzlich umso mehr durch eine parallelisierte Ausführung beschleunigt werden kann, je stärker der in jedem Simulationsschritt benötigte Rechenaufwand ist und je schwächer das Simulationsmodell innerlich gekoppelt ist. Das ideale Simulationsmodells für die Ausführung auf n Recheneinheiten ist demnach ein Modell, das aus n separaten Teilmodellen besteht, deren jeweilige Berechnung sehr aufwendig ist.

Aber selbst das Vorliegen eines geeigneten Simulationsmodells ist noch kein Garant für einen hohen Geschwindigkeitsgewinn. Erst durch eine gute Verteilung des Simulationsmodells auf die verfügbaren Recheneinheiten kann eine Simulationsbeschleunigung erzielt werden.

Dabei ist das Identifizieren einer derartigen guten Verteilung mitnichten trivial. In den meisten durchgeführten Experimente konnte zwar die in der parallelen Simulationsliteratur häufig anzutreffende Empfehlung, räumlich und/oder modellzeitlich schwach gekoppelte Modellbestandteile am ehesten auf separate Recheneinheiten zu verteilen, bestätigt werden. In einer Minderheit der durchgeführten Experimente erwies sich aber genau dieser Verteilungsansatz als kontraproduktiv. Insofern muss im praktischen Anwendungsfall bei einem realen Simulationsmodell eine Einzelfallprüfung als unumgänglich bezeichnet werden.

Insbesondere besteht bei einer schlechten Verteilung die Gefahr, dass aufgrund des parallelisierungsbedingten, zusätzlichen Rechenaufwandes der optimistisch-parallelen Simulationsausführung diese mehr Ausführungszeit benötigen kann als eine sequentielle. Auch dieser Umstand konnte in den durchgeführten Experimenten bestätigt werden.

7.3 Ausblick

Nachnutzung der generischen Prozessimplementation

Der für die Prozessimplementation gewählte Coroutinen/Continuation-Implementationsansatz auf Basis von JAVAFLOW ist nicht nur für die Verwendung in optimistisch-parallelen, sondern auch sequentiellen oder konservativ-parallelen Simulationskernen auf Java-Basis geeignet. In Anhang C.1 werden die Vorteile und Nachteile einer derartigen Prozessimplementation, vor allem gegenüber dem üblichen Java-Implementationsansatz auf Basis zwangsserialisierter Threads, kurz ausgeführt.

Zusammengefasst ermöglicht eine Prozessimplementation auf JAVAFLOW-Basis eine höhere Anzahl von Prozessen, dafür ist die konkrete Umsetzung etwas aufwendiger. Bezüglich der Ausführungsgeschwindigkeit musste die überraschende Feststellung getroffen werden, dass die jeweils schnellere Prozessimplementation maßgeblich vom verwendeten Betriebssystem abhängt.

Erst kurz vor Abschluss dieser Dissertationsschrift wurde der Autor aufgrund der thematisch zugehörigen Veröffentlichung [Kun08] von einem Mitarbeiter der TU-Delft kontaktiert. Dieser reimplementiert derzeit die generische Prozessimplementation des konservativ-parallel arbeitenden Simulationskerns der dort entwickelten Simulationsbibliothek DSOL [JLV02]. Dabei verwendet er ebenfalls die Coroutinen/Continuation-Implementation von JAVAFLOW als Basis. Eine Veröffentlichung seiner Erfahrungen während der Implementation als auch der erzielten Geschwindigkeitsgewinne gegenüber der derzeitigen interpretativen Prozessimplementation ist in näherer Zukunft zu erwarten.

Weiterentwicklung von MYTIMEWARP

Wie in Abschnitt 5.6 beschrieben wurde, lässt sich die aktuelle Implementation von MYTIMEWARP am ehesten als prototypisch bezeichnen. Bevor MYTIMEWARP in praktischen Szenarien eingesetzt werden kann, müssen noch einige fehlende Komponenten bzw. Funktionalitäten implementiert werden.

Die beiden größten Mängel im aktuellen Entwicklungsstand stellen sicherlich die nicht existierende Berechnung der GVT und die ebenfalls fehlende, auf der GVT-Berechnung basierende Freigabe obsoleter Prozesszustände (fossil collection) dar. Infolgedessen wächst derzeit der Speicherbedarf während einer Simulationsausführung kontinuierlich an, was wiederum dazu führt, dass nur kleine Simulationsmodelle verwendet und/oder kurze Modellzeiträume betrachtet werden können. Daher stellt eine Vervollständigung des Simulationskerns um diese beiden Aspekte eine der dringendsten Aufgaben bei einer Weiterentwicklung von MYTIMEWARP dar.

Ein weiteres Problem in der derzeitigen Implementation sind die spärlichen Möglichkeiten, während eines Simulationslaufes Informationen über den aktuellen Modellzustand auszugeben bzw. anderweitig zu konservieren. Gegenwärtig existiert ausschließlich die Möglichkeit, einfache Zeichenketten mit der jeweils aktuellen Modellzeit zusammen in die Standardausgabe zu schreiben. Da auf dieser Basis detaillierte Auswertungen eines Simulationslaufes zwar möglich, aber sehr aufwendig sind, sollten bei einer angedachten Anwendung von MYTIMEWARP in der Praxis Hilfsfunktionen zum Sammeln und Auswerten von statistischen Daten, analog zu anderen Simulationsbibliotheken, implementiert werden.

Bei dieser Gelegenheit sollte die Simulationsbibliothek auch um zusätzliche Hilfsmittel zur Simulationsmodellerstellung erweitert werden. Eine Möglichkeit der Nutzung generisch vorimplementierter Modellkomponenten wie Warteschlangen verschiedenen Typs, Semaphoren usw. analog zu existierenden Simulationsbibliotheken würde nicht nur die Modellerstellung deutlich erleichtern. Auch die Akzeptanz von MYTIMEWARP für praktische Anwendungen würde davon profitieren.

Simulationen realer Szenarien

Eine derartig weiterentwickelte MYTIMEWARP-Implementation sollte dann auch mit Simulationsmodellen realer Szenarien getestet werden, deren zugrundeliegende Problematik von sich aus hinreichend kompliziert bzw. rechenaufwendig sind. So würde der praktische Nutzen der parallelen Simulationsausführung im Allgemeinen und deren konkrete Umsetzung in MYTIMEWARP im Speziellen noch deutlicher werden.

Eine derartige Simulationsimplementation entsteht derzeit im Rahmen einer studentischen Arbeit am Lehrstuhl Systemanalyse der Humboldt-Universität zu Berlin. Dabei handelt es ich um ein Simulationsszenario aus dem Gebiet der Erdbebenforschung, speziell der Erdbebenausbreitung und -frühwarnung, das auf Basis der prototypischen MYTIMEWARP-Implementation umgesetzt wird, wobei die bisheri-

gen Erfahrungen bei der Implementation und frühen Simulationsläufen vielversprechend sind.

ANHANG A

IMPLEMENTIERTE SIMULATIONSMODELLE

In diesem Anhang werden die Quelltexte mehrerer, einfacher Simulationsmodelle ungekürzt wiedergegeben. Wenn bereits eine Beschreibung des jeweiligen Szenarios in den vorherigen Kapiteln vorhanden ist, wird diese nur referenziert, ansonsten erfolgt an Ort und Stelle eine Kurzbeschreibung.

A.1 Ping-Pong (ereignisorientiert)

Die beiden folgenden Quelltexte stellen die Implementation eines einfachen Ping-Pong-Szenarios dar. Dabei wurde das Simulationsmodell ereignisorientiert modelliert: Es existiert genau ein Ereignistyp, implementiert in der Klasse *PingPongEvent*, von dem zur Laufzeit exakt zwei Exemplare (ein „Ping“- und ein „Pong“-Ereignis) instantiiert werden (in dieser Implementation werden die beiden einmal erzeugten Ereignisse wiederverwendet, d. h., statt in jedem Simulationsschritt ein neues Ereignis des entsprechenden Ereignistyps zu instantiieren, werden die bereits existierenden Ereignisse einfach erneut in den Ereigniskalender eingetragen).

Die Implementation der Simulationsinitialisierung und -steuerung erfolgt in der Klasse *PingPongEventSimulation*. Hier werden zwei LPs sowie zwei Ereignisse instantiiert, wobei die beiden Ereignisse auch gleich mit jeweils einem LP verbunden werden. Nach der gegenseitigen Bekanntmachung der Ereignisse wird die Simulation gestartet. Sobald diese beendet ist, werden die Simulationsergebnisse ausgegeben.

Quelltext A.1: Die Klasse *PingPongEvent*

```
1 package pingpong;
2
3 import simulator.Event;
4 import simulator.LogicalProcess;
```

```
5
6  /**
7   * Implementation eines Ping-/Pong-Ereignisses.
8   * @author Andreas Kunert
9   */
10 public class PingPongEvent extends Event {
11
12    /** das jeweils andere Ereignis */
13    private Event otherEvent;
14
15    /**
16     * Konstruktor
17     * @param logicalProcess der das Ereignis verarbeitende LP
18     * @param id Identifikationsnummer des Ping-Pong-Ereignisses
19     */
20    public PingPongEvent(LogicalProcess logicalProcess, int id) {
21       super(logicalProcess, "PPE#" + id);
22    }
23
24    /**
25     * Modellverhalten bei einem Ping/Pong-Ereignis.
26     */
27    @Override
28    public void eventRoutine() {
29       output(name + " \t\t(" + logicalProcess + ")");
30       // Eintragen des anderen Ereignisses im Ereigniskalender eintragen
31       scheduleEvent(otherEvent, logicalProcess.getCurrentTime().time + 10);
32    }
33
34    /**
35     * legt den jeweils andere Ereignistyp fest ("Pong" zu "Ping" bzw. umgekehrt)
36     * @param otherEvent der andere Ereignistyp
37     */
38    public void setOtherEvent(Event otherEvent) {
39       this.otherEvent = otherEvent;
40    }
41
42 }
```

Quelltext A.2: Die Klasse *PingPongEventSimulation*

```
1  package pingpong;
2
3  import simulator.LogicalProcess;
4  import simulator.Simulation;
5
6  /**
7   * Einfache Implementation einer ereignisorientiert modellierten Ping-Pong-Simulation.
8   * @author Andreas Kunert
9   */
10 public class PingPongEventSimulation {
11
12    /**
13     * Startmethode der Simulation.
```

```
14      * @param argv nicht benutzt
15      */
16     public static void main(String[] argv) {
17
18         // Simulation anlegen und Simulationsendzeit festlegen
19         long endTime = 1000;
20         Simulation simulation = new Simulation(endTime);
21
22         // Logische Prozesse anlegen und mit der Simulation verbinden
23         LogicalProcess lp0 = new LogicalProcess(simulation);
24         LogicalProcess lp1 = new LogicalProcess(simulation);
25
26         // Ping-Pong-Ereignisse anlegen und mit dem LP verbinden
27         PingPongEvent ppe0 = new PingPongEvent(lp0, 0);
28         PingPongEvent ppe1 = new PingPongEvent(lp1, 1);
29
30         // Ping-Pong-Ereignisse einander bekannt machen
31         ppe0.setOtherEvent(ppe1);
32         ppe1.setOtherEvent(ppe0);
33
34         // Startereignis ("Ping") zur Modellzeit 0 festlegen
35         lp0. setInitialEvent (ppe0, 0);
36
37         // Simulation starten
38         simulation.startSimulation ();
39
40         // Serialisierte Ausgabe ausgeben
41         simulation.dumpSerializedOutput();
42     }
43
44 }
```

A.2 Ping-Pong (prozessorientiert)

Die folgenden Quelltexte stellen eine prozessorientiert modellierte Implementation des Ping-Pong-Szenarios dar. Anstatt wie im vorherigen Beispiel die Ereignisse des „Ping“ bzw. „Pong“ zu modellieren, werden die beteiligten Spieler als Prozesse (in Form von Instanzen der Klasse *PingPongProcess*) umgesetzt.

Die Steuerklasse *PingPongProcessSimulation* hat einen fast identischen Aufbau wie die bereits vorgestellte Klasse *PingPongEventSimulation* aus der ereignisorientierten Implementationsvariante. Der einzige Unterschied besteht in der Instantiierung von Prozessen (anstatt von Ereignissen) und der Festlegung der initialen Aktivierung (anstelle der Identifikation eines Startereignisses).

Quelltext A.3: Die Klasse *PingPongProcess*

```
1 package pingpong;
2
3 import simulator.LogicalProcess;
4 import simulator.SimProcess;
```

```
5
6  /**
7   * Implementation eines Ping-/Pong-Prozesses.
8   * @author Andreas Kunert
9   */
10 public class PingPongProcess extends SimProcess {
11
12    /** der jeweils andere Prozess */
13    private simulator.SimProcess otherProcess;
14
15    /**
16     * Konstruktor
17     * @param logicalProcess der den Prozess ausführende LP
18     * @param id Identifikationsnummer des Ping-Pong-Prozesses
19     */
20    public PingPongProcess(LogicalProcess logicalProcess, int id) {
21       super(logicalProcess, "PPP#" + id);
22    }
23
24    /**
25     * Lebenszyklusmethode des Ping/Pong-Prozesses.
26     */
27    @Override
28    public void run() {
29       while(true) {
30          output(name + " \t\t(" + logicalProcess + ")");
31          // Aktivierung des anderen Prozesses im Ereigniskalender eintragen
32          scheduleProcess(otherProcess, logicalProcess.getCurrentTime().time + 10);
33          // Warten auf Reaktivierung durch den anderen Prozess
34          suspend();
35       }
36    }
37
38    /**
39     * legt den jeweils anderen Prozess fest ("Pong" zu "Ping" bzw. umgekehrt)
40     * @param otherProcess der andere Prozess
41     */
42    public void setOtherProcess(SimProcess otherProcess) {
43       this.otherProcess = otherProcess;
44    }
45
46 }
```

Quelltext A.4: Die Klasse *PingPongProcessSimulation*

```
1  package pingpong;
2
3  import simulator.LogicalProcess;
4  import simulator.Simulation;
5
6  /**
7   * Einfache Implementation einer prozessorientiert modellierten Ping-Pong-Simulation.
8   * @author Andreas Kunert
9   */
```

```
10 public class PingPongProcessSimulation {
11
12   /**
13    * Startmethode der Simulation.
14    * @param argv nicht benutzt
15    */
16   public static void main(String[] argv) {
17
18     // Simulation anlegen und Simulationsendzeit festlegen
19     long endTime = 1000;
20     Simulation simulation = new Simulation(endTime);
21
22     // Logische Prozesse anlegen und mit der Simulation verbinden
23     LogicalProcess lp0 = new LogicalProcess(simulation);
24     LogicalProcess lp1 = new LogicalProcess(simulation);
25
26     // Ping-Pong-Prozesse anlegen und mit dem LP verbinden
27     PingPongProcess ppp0 = new PingPongProcess(lp0, 0);
28     PingPongProcess ppp1 = new PingPongProcess(lp1, 1);
29
30     // Ping-Pong-Prozesse einander bekannt machen
31     ppp0.setOtherProcess(ppp1);
32     ppp1.setOtherProcess(ppp0);
33
34     // "Ping"-Prozess initial aktivieren (zur Modellzeit 0)
35     ppp0. initialActivate (0);
36
37     // Simulation starten
38     simulation.startSimulation ();
39
40     // Serialisierte Ausgabe ausgeben
41     simulation.dumpSerializedOutput();
42   }
43 }
```

A.3 Dining Philosophers

Die beiden folgenden Quelltexte der Klassen *PhilosopherSimulation* und *PhilosopherProcess* stellen die gesamte Implementation der in Kapitel 5 beschriebenen Simulation des Philosophenproblems nach Edgar Dijkstra dar.

Quelltext A.5: Die Klasse *PhilosopherProcess*

```
1 package philosophers;
2
3 import java.security.MessageDigest;
4
5 import org.apache.log4j.Logger;
6
7 import simulator.LogicalProcess;
8 import simulator.Resource;
9 import simulator.SimProcess;
```

```
10
11 /**
12  * Prozess, der das Verhalten eines Philosophen im Philosophenproblem nachbildet.
13  * @author Andreas Kunert
14  * @see PhilosopherSimulation
15  */
16 public class PhilosopherProcess extends SimProcess {
17
18    /** Statischer Logger */
19    private static final Logger logger = Logger.getLogger(PhilosopherProcess.class.getName());
20
21    /**
22     * konstanter Parameter für die künstliche Verzögerung in jedem Wartezustand, wird
23     * wird vom Simulationshauptprogramm gesetzt
24     * @see PhilosopherProcess#spendTime()
25     */
26    private final int delayParameter;
27    /**
28     * Art der künstlichen Verzögerung.
29     * 0 .. Rekursive Fibonaccizahlenberechnung
30     * 1 .. Berechnung des n-ten MD5-Hashwertes
31     */
32    private final int kindOfDelay;
33    /** die linke Gabel (als boolesche Ressource), Zuordnung erfolgt im Konstruktor */
34    private final Resource<Boolean> leftFork;
35    /** die rechte Gabel (als boolesche Ressource), Zuordnung erfolgt im Konstruktor */
36    private final Resource<Boolean> rightFork;
37
38    /**
39     *  Ungenutzte Variable. Dient dazu, dass die künstlichen Verzögerungen nicht
40     *  mangels Nutzen herausoptimiert werden.
41     */
42    public int dummyInt;
43    /**
44     *  Ungenutzte Variable. Dient dazu, dass die künstlichen Verzögerungen nicht
45     *  mangels Nutzen herausoptimiert werden.
46     */
47    public byte[] hash;
48
49
50    /**
51     * Konstruktor
52     * @param logicalProcess der logische Prozess, auf dem der Philosophenprozess ausgeführt wird
53     * @param number eindeutige Identifikationsnummer des Philosophenprozesses
54     * @param leftFork die linke Gabel (als Ressource)
55     * @param rightFork die rechte Gabel (als Ressource)
56     * @param delayParameter konstanter Parameter für die künstliche Verzögerung
57     * @param kindOfDelay Art der künstlichen Verzögerung
58     */
59    public PhilosopherProcess(
60          LogicalProcess logicalProcess, int number,
61          Resource<Boolean> leftFork, Resource<Boolean> rightFork,
62          int delayParameter, int kindOfDelay) {
63
```

```
64      super(logicalProcess, "Philosopher " + number);
65      this.leftFork = leftFork;
66      this.rightFork = rightFork;
67      this.delayParameter = delayParameter;
68      this.kindOfDelay = kindOfDelay;
69    }
70
71    /**
72     * Lebenszyklusmethode eines Philosophen.
73     * Ein Philosoph versucht nacheinander in Besitz beider anliegender Gabeln zu gelangen.
74     * Bei Erfolg verfällt er in einen Wartezustand (Essen), legt die Gabeln zurück und verfällt
75     * in einen zweiten Wartezustand (Denken).
76     * Sollte hingegen eine der Gabeln nicht verfügbar gewesen sein, so verfällt er gleich in den
77     * zweiten Wartezustand.
78     * Das eigentliche Warten übernimmt die Methode <code>spendTime(int)</code>.
79     */
80    @Override
81    public void run() {
82
83      logger.info(makeLogMessage("starting " + this));
84      output(name + " ... starting");
85
86      // Ewig (bis zum Simulationsende)
87      while (true) {
88        // nehmen der linken Gabel
89        if (getResource(leftFork)) {
90          setResource(leftFork, false);
91          logger.info(makeLogMessage("takes left"));
92          output(name + " ... takes left");
93
94          // nehmen der rechten Gabel
95          if (getResource(rightFork)) {
96            setResource(rightFork, false);
97            logger.info(makeLogMessage("takes right"));
98            output(name + " ... takes right");
99
100           // warten (essen)
101           logger.info(makeLogMessage("eating"));
102           output(name + " ... eating");
103           spendTime();
104
105           // zurücklegen der rechten Gabel
106           setResource(rightFork, true);
107           logger.info(makeLogMessage("puts right"));
108           output(name + " ... puts right");
109         }
110
111         // zurücklegen der linken Gabel
112         setResource(leftFork, true);
113         logger.info(makeLogMessage("puts left"));
114         output(name + " ... puts left");
115       }
116
117       // warten (denken)
```

```
118             logger.info(makeLogMessage("thinking"));
119             output(name + " ... thinking");
120             spendTime();
121         }
122     }
123
124     /**
125      * das Verweilen während sich der Prozess in einem der beiden Wartezustände (Essen, Denken)
126      * befindet. Das Warten im Sinne der Simulation übernimmt die Methode <code>sleep</code>.
127      * Zu Testzwecken wird vorher die potentiell rechenaufwendige, rekursive Berechnung einer
128      * Fibonaccizahl durchgeführt.
129      */
130     protected void spendTime() {
131         // Starten der künstlichen Verzögerung
132         // (Zuweisung in eine Variable, damit der Funktionsaufruf nicht wegoptimiert wird)
133         if (delayParameter>0) {
134
135             if (kindOfDelay == 0) {
136                 dummyInt = fibo(delayParameter);
137             }
138             else if (kindOfDelay == 1) {
139                 String string = name;
140                 try {
141                     MessageDigest md = MessageDigest.getInstance("MD5");
142                     md.update(string.getBytes());
143                     hash = md.digest();
144                     for (int i=1; i<delayParameter; i++) {
145                         md.update(hash);
146                         hash = md.digest();
147                     }
148                 }
149                 catch (Exception e) { logger.fatal(e); System.exit(-1); }
150             }
151             else {
152                 logger.fatal("unknown kind of delay requested");
153                 System.exit(-1);
154             }
155         }
156
157
158
159         // Warten im Sinne der Fortführung der Modellzeit
160         sleep(10);
161     }
162
163     /**
164      * berechnet rekursiv die n-te Fibonaccizahl.
165      * @param n Konstante, zu der die Fibonaccizahl berechnet werden soll
166      * @return die n-te Fibonaccizahl
167      */
168     protected int fibo(int n) {
169         if (n < 2) return 1;
170         else return fibo(n-1)+fibo(n-2);
171     }
```

```
172
173     /**
174      * Hilfsmethode zur Anreicherung von Logausgaben um Daten bezüglich des aktuellen Prozesses
175      * @param string das die Logausgabe verursachende Ereignis
176      * @return String, der das übergebene Ereignis sowie die aktuelle Modellzeit, den Namen des
177      *           Prozesse und den logischen Prozess enthält
178      */
179     public String makeLogMessage(String string) {
180         return "(" + getCurrentTime() + ") " + name + " (" + logicalProcess + ") " + string;
181     }
182
183     /**
184      * überschriebene clone()-Methode, die vom optimistischen Simulationskern für das fortlaufende
185      * Anlegen von Prozesskopien dieses Prozesses benutzt wird.
186      */
187     @Override
188     public PhilosopherProcess clone() {
189         return (PhilosopherProcess) super.clone();
190     }
191
192 }
```

Quelltext A.6: Die Klasse *PhilosopherSimulation*

```
1  package philosophers;
2
3  import java.util.Vector;
4
5  import org.apache.log4j.Logger;
6  import org.apache.log4j.PropertyConfigurator;
7
8  import simulator.LogicalProcess;
9  import simulator.Resource;
10 import simulator.Simulation;
11 import benchmark.Timer;
12
13 /**
14  * Simulation einer Variante des Philosophenproblems von Dijkstra.
15  * @author Andreas Kunert
16  */
17 public class PhilosopherSimulation {
18
19     /** Statischer Logger */
20     private final static Logger logger = Logger.getLogger(PhilosopherSimulation.class.getName());
21
22     /** Anzahl der Philosophen */
23     private static int numberOfPhilosophers = 12;
24     /** Anzahl der logischen Prozesse (= Anzahl der parallel laufenden Threads) */
25     private static int numberOfLPs = 3;
26     /** Simulationsendzeit (Simulation läuft von 0 bis endTime) */
27     private static long endTime = 100;
28     /** konstanter Parameter für die künstliche Verzögerung
29      *  @see PhilosopherProcess#spendTime()
30      */
```

```
31    private static int delayParameter = 0;
32    /** Art der Verteilung von Philosophenprozessen auf logische Prozesse.
33     * 0 .. Kreissegmentverteilung
34     * 1 .. Round-Robin-Verteilung
35     */
36    private static int roundRobin = 0;
37    /**
38     * Art der künstlichen Verzögerung.
39     * 0 .. Rekursive Fibonaccizahlenberechnung
40     * 1 .. Berechnung des n-ten MD5-Hashwertes
41     */
42    private static int kindOfDelay = 1;
43
44    /** Vektor der Gabeln als boolesche Ressourcen */
45    static Vector<Resource<Boolean>> forks = new Vector<Resource<Boolean>>();
46    /** Array der Philosophenprozesse */
47    static PhilosopherProcess[] philosophers;
48    /** Array der logischen Prozesse */
49    static LogicalProcess[] lps;
50
51    /**
52     * Startmethode der Simulation.
53     * @param argv Simulationsparameter
54     */
55    public static void main(String[] argv) {
56       // Logger konfigurieren
57       PropertyConfigurator.configure("log4j.properties");
58
59       // Timer anlegen
60       Timer t1 = new Timer(); // Bruttolaufzeit (gesamtes Programm)
61       Timer t2 = new Timer(); // Nettolaufzeit (reine Simulation)
62       t1.start();
63
64       // Simulationsparameter auswerten
65       // (wenn Parameter fehlen gelten obige Defaultwerte)
66       if (argv.length > 0) numberOfPhilosophers = Integer.parseInt(argv[0]);
67       System.err.println("numberOfPhilosophers: " + numberOfPhilosophers);
68       philosophers = new PhilosopherProcess[numberOfPhilosophers];
69       if (argv.length > 1) numberOfLPs = Integer.parseInt(argv[1]);
70       System.err.println("numberOfLPs: " + numberOfLPs);
71       lps = new LogicalProcess[numberOfLPs];
72       if (argv.length > 2) endTime = Integer.parseInt(argv[2]);
73       System.err.println("endTime: " + endTime);
74       if (argv.length > 3) delayParameter = Integer.parseInt(argv[3]);
75       System.err.println("delayParameter: " + delayParameter);
76       if (argv.length > 4) roundRobin = Integer.parseInt(argv[4]);
77       System.err.println("roundRobin: " + roundRobin);
78       if (argv.length > 5) kindOfDelay = Integer.parseInt(argv[5]);
79       System.err.println("kindOfDelay: " + kindOfDelay);
80
81       // Simulationsumgebung anlegen
82       Simulation simulation = new Simulation(endTime);
83
84       // Ressourcenobjekte der Gabeln instantiieren und initialisieren
```

```
85      logger.debug(" initializing  forks");
86      for (int i = 0; i < numberOfPhilosophers; i++) {
87          Resource<Boolean> fork = new Resource<Boolean>();
88          fork. setInitialValue (true);
89          forks.add(fork);
90      }
91
92      // Logische Prozesse instantiieren und mit der Simulation verbinden
93      logger.debug(" initializing  LPs");
94      for (int i = 0; i < numberOfLPs; i++) {
95          lps[i] = new LogicalProcess(simulation);
96      }
97
98      // Philosophenprozesse instantiieren und dabei:
99      // - diese je nach Verteilung den logischen Prozessen zuordnen
100     // - die benachbarten Gabeln den Philosophenprozessen zuteilen
101     logger.debug(" initializing  philosophers");
102     for (int i = 0; i < numberOfPhilosophers; i++) {
103         if (roundRobin == 1) {
104             philosophers[i] = new PhilosopherProcess(
105                     lps[i%numberOfLPs], i,
106                     forks.elementAt(i), forks.elementAt((i + 1) % numberOfPhilosophers),
107                     delayParameter, kindOfDelay);
108         }
109         else if (roundRobin == 0){
110             philosophers[i] = new PhilosopherProcess(
111                     lps[(int) ((i-i%((double) numberOfPhilosophers/numberOfLPs))
112                     /((double) numberOfPhilosophers/numberOfLPs))],
113                     i, forks.elementAt(i), forks.elementAt((i + 1) % numberOfPhilosophers),
114                     delayParameter, kindOfDelay);
115         }
116         else {
117             logger. fatal ("unknown distribution method");
118             System.exit(-1);
119         }
120         philosophers[i]. initialActivate (0);
121     }
122
123     // Simulation starten und dabei die Laufzeit mit dem Timer t2 messen
124     logger. info ("user starts simulation");
125     t2. start ();
126     simulation. startSimulation ();
127     t2.stop ();
128     logger. info ("user sees finished simulation");
129
130     // Serialisierte Ausgabe erstellen und anzeigen
131     logger.debug("dumping serializedOutput");
132     simulation.dumpSerializedOutput();
133
134     // Simulation beenden und als letzte Aktion Timer t1 (Bruttolaufzeit) stoppen
135     logger. info ("ending PhilosopherSimulation");
136     t1.stop ();
137
138     // Ausgabe der Simulations- und Messergebnisse
```

```
139         System.out.println(
140                 "Conf:" + numberOfPhilosophers + "," + numberOfLPs + "," + endTime + ","
141                 + delayParameter + "," + roundRobin + " Result: " + t2.result()
142                 + "  TimeWarps: " + simulation.getOverallNumberOfTimeWarps());
143     }
144
145 }
```

ANHANG B

VERWANDTE ARBEITEN

B.1 Desmo-J

DESMO-J (*Discrete Event Simulation and MOdelling in Java*) ist eine Simulationsbibliothek, die am Arbeitsbereich *Angewandte und sozialorientierte Informatik* der Universität Hamburg entstanden ist und erstmals in [LP99] beschrieben wurde. Dabei basiert DESMO-J maßgeblich auf einer 1987–1989 am selben Arbeitsbereich entstandenen Simulationsbibliothek namens DESMO (*Discrete Event Simulation in MOdula 2*). Diese stellt wiederum eine erweiterte Reimplementation der erstmals 1979 von Birtwistle vorgestellten Simulationsbibliothek DEMOS [Bir81] dar. Neben DESMO-J sind an der Universität Hamburg noch weitere Portierungen von DESMO in andere Programmiersprachen wie Smalltalk, Oberon, C++ und Delphi entstanden.

DESMO-J ermöglicht die Implementation diskreter Simulationen, wobei sowohl ereignisorientierte als auch prozessorientierte Simulationsmodelle unterstützt werden. Zur Umsetzung der Prozesse prozessorientierter Simulationsmodelle verwendet DESMO-J im Simulationskern asymmetrische Coroutinen, die wiederum auf der Basis der in Abschnitt 4.6 diskutierten Variante der zwangsserialisierten Threads implementiert wurden. Demzufolge wird jeder Prozess während eines Simulationslaufes in einem separaten Java-Thread ausgeführt. Da DESMO-J ausschließlich sequentielle Simulationen unterstützt, ist es nicht durch die in Abschnitt 4.6 diskutierten Probleme durch die fehlende Mehrfachfortsetzbarkeit der dabei realisierten Continuations betroffen. Ein Nachteil der gewählten Thread-Implementation besteht jedoch darin, dass die maximal mögliche Anzahl von Prozessen in einem Simulationsmodell durch die maximal mögliche Anzahl von Java-Threads begrenzt wird (siehe auch Anhang C.1).

Eine hervorzuhebende Eigenschaft von DESMO-J ist der Umfang der zur Verfügung stehenden Dokumentation. Wie bei nahezu allen Simulationsbibliotheken üblich, besteht diese zunächst aus einer detaillierten Beschreibung der Programmier-

schnittstellen (API) sowie einer Sammlung von vollständig implementierten und ausführlich dokumentierten Beispielsimulationen. Darüber hinaus existiert jedoch zusätzlich ein vorbildliches Web-Tutorial für DESMO-J. In diesem werden anhand eines ausführlich beschriebenen Beispielszenarios zuerst die Modellerstellung (sowohl prozessorientiert als auch ereignisorientiert) und anschließend die Simulationsimplementation in Desmo-J in verschiedenen Varianten erläutert. Des Weiteren wird DESMO-J als zugrundeliegende Simulationsbibliothek für alle praktischen Beispiele im Simulationsgrundlagenbuch *Java Simulation Handbook* [PK05] verwendet.

B.2 JAVASIMULATION und JDISCO

Bei JAVASIMULATION [Hel00] handelt es sich um eine sequentielle Simulationsbibliothek, die an der Roskilde University entstanden ist. Wie DESMO-J basiert auch JAVASIMULATION auf der SIMULA-Simulationsbibliothek DEMOS und verwendet zwangsserialisierte Threads zur Implementation von Coroutinen für Prozesse. Im Gegensatz zu DESMO-J wurde jedoch bei JAVASIMULATION der ursprüngliche DEMOS-Ansatz, Prozesse auf Basis symmetrischer Coroutinen zu realisieren, beibehalten.

Infolgedessen sind zur Laufzeit einer Simulation die jeweils aktiven Prozessthreads selbst dafür zuständig, stets unmittelbar vor ihrer eigenen Unterbrechung den laut Ereigniskalender nachfolgenden Prozessthread zu aktivieren. Bei JAVASIMULATION besteht dabei eine Besonderheit darin, dass der Ereigniskalender nicht als separate Datenstruktur realisiert wurde. Stattdessen wurde jeder Prozess mit zwei zusätzlichen Variablen ausgestattet, die Referenzen auf den unmittelbaren Vorgänger- und Nachfolgerprozess enthalten. Dadurch entfällt zur Laufzeit das Ermitteln des nachfolgenden Prozessthreads, dafür ist der Aufwand beim Einfügen von Prozessaktivierungen in diesen virtuellen Ereigniskalender etwas höher.

Ein Nachteil der Wahl symmetrischer Coroutinen fällt bei der Betrachtung ereignisorientierter Simulationsmodelle auf. In Simulationskernen, die auf asymmetrischen Coroutinen basieren, kann das Auftreten eines Ereignisses durch ein einfaches Aufrufen der entsprechenden Ereignisroutine umgesetzt werden. Bei Simulationskernen hingegen, die auf der Steuerungsweitergabe von Coroutine zu Coroutine basieren, fehlt beim Auftreten eines Ereignisses eine zu aktivierende Coroutine.

Im Gegensatz zu anderen Simulationsbibliotheken auf Basis symmetrischer Coroutinen (z. B. ODEMX [Ger03]) wird dieses Problem in JAVASIMULATION vollständig ignoriert. Es gibt in der Programmierschnittstelle kein Konzept eines universellen Ereignisses, so dass JAVASIMULATION als Simulationsbibliothek angesehen werden kann, die ausschließlich prozessorientierte Simulationsmodelle unterstützt. JAVASIMULATION besitzt zwar (primär zu Demonstrationszwecken) auch einen separaten Simulationskern für ereignisorientierte Simulation, dieser unterstützt jedoch ausschließlich ereignisorientierte Simulationsmodelle und ist vom prozessorientierten Simulationskern vollständig getrennt. Auch die Beispielimplementation eines ereignisorientierten Simulationsmodells im Anhang von [Hel00] ist nur dadurch möglich,

dass alle auftretenden Ereignisse als Prozesse realisiert wurden, d. h., das ereignisorientierte Simulationsmodell wurde bei der Implementation in ein prozessorientiertes überführt.

Die Simulationsbibliothek JDISCO, die hier nur der Vollständigkeit halber erwähnt werden soll, ist eine Erweiterung von JAVASIMULATION, mittels derer auch zeitkontinuierliche Simulationsmodelle implementiert werden können. Zu diesem Zweck wurde eine spezielle Klasse von Prozessen, die sogenannten zeitkontinuierlichen Prozesse eingefügt. Innerhalb dieser können Differentialgleichungen beschrieben werden, deren zur Laufzeit numerisch bestimmte Lösungen als Eintrittszeitpunkte von Prozessaktivierungen (sowohl kontinuierlicher als auch diskreter Prozesse) verwendet werden können.

B.3 ODEM und ODEMX

Bei ODEM (*Object-oriented Discrete Event Modelling*, [FA96]) handelt es sich um eine sequentielle Simulationsbibliothek, die am Lehrstuhl Systemanalyse des Instituts für Informatik der Humboldt-Universität zu Berlin entstanden ist und die ebenfalls nach dem Vorbild der SIMULA-Bibliothek DEMOS entwickelt wurde. Im Unterschied zu den bisher betrachteten Simulationsbibliotheken wurde der Simulationskern von ODEM jedoch in C++ umgesetzt und auch die Simulationsmodelle müssen in dieser Programmiersprache implementiert werden. Dabei unterstützt ODEM neben zeitdiskreten auch zeitkontinuierliche Simulationsmodelle.

Analog zu DEMOS und JAVASIMULATION basiert die Prozessimplementation von ODEM auf symmetrischen Coroutinen. Diese wiederum wurden auf Basis der in Abschnitt 4.6 beschriebenen Verfahren Registersatzsicherung und Stackmanipulation implementiert. Bedingt durch die symmetrischen Coroutinen erfolgt in ODEM die Steuerungsweitergabe direkt von Prozess zu Prozess, wobei jedoch im Unterschied zu JAVASIMULATION der Ereigniskalender als separate Datenstruktur umgesetzt wurde.

ODEMX ist eine Weiterentwicklung und partielle Reimplementation von ODEM, die im Rahmen einer Diplomarbeit [Ger03] entstanden ist. In den ersten Versionen von ODEMX standen vor allem die Überarbeitung der Softwarearchitektur von ODEM und die Einführung bzw. stärkere Verwendung neuer Konzepte der Programmiersprache C++ im Vordergrund. Inzwischen liegt der Schwerpunkt der Weiterentwicklung in der Erweiterung der unterstützten Modellierungskonzepte. Erwähnenswert ist hierbei insbesondere die in ODEM nicht vorhandene, native Unterstützung ereignisorientierter Simulationsmodelle.

B.4 JIST

Im Gegensatz zu den bisher betrachteten Simulationsbibliotheken geht JIST (*Java in Simulation Time*, [Bar04]) einen völlig anderen Weg der Simulationsrealisierung.

Die Grundidee besteht darin, durch eine Änderung der Ausführungssemantik von Java die Ausführung normaler Java-Programme nachträglich mit dem Konzept einer Modellzeit zu verbinden. Dadurch wird eine Simulationsmodellausführung ohne sichtbaren Simulationskern möglich. Das Simulationsmodell wird teilweise selbst um die Funktionalitäten eines Simulationskerns angereichert.

Die Umsetzung der Modifikation der Ausführungssemantik basiert dabei auf dem gleichen Ansatz, den auch die in Abschnitt 4.7 beschriebenen Continuation-Implementationen JAVAFLOW und RIFE/CONTINUATIONS verwenden: der nachträglichen Modifikationen der kompilierten Java-Programme durch einen Bytecode-Rewriter.

Die Grundlage aller Simulationsmodelle in JIST sind Klassen, die die spezielle, leere Java-Schnittstelle *JistAPI.Entity* implementieren. Es sind ausschließlich derartig „markierte“ Klassen, die später vom Bytecode-Rewriter modifiziert werden. Die Methoden dieser Klasse entsprechen weitestgehend den Ereignisroutinen ereignisorientierter Simulationsmodelle. Dies macht sich vor allem dadurch bemerkbar, dass der Aufruf einer dieser Methoden mitnichten der Ausführung derselbigen, sondern der Eintragung des zugehörigen Ereignisses (für das keine separate Datenstruktur vorgesehen ist) im Ereigniskalender entspricht.

Interessant ist dabei das Problem der Festlegung des Eintrittszeitpunktes ohne dessen explizite Angabe als Parameter gelöst worden. Zunächst kann eine aktive Ereignisroutine in JIST zum Modellzeitpunkt t_0 nur Ereignisse zum selben Zeitpunkt t_0 im Ereigniskalender eintragen, indem sie die zugehörige Ereignisroutinen direkt als Java-Methoden aufruft. Wie bereits beschrieben, führt dies *nicht* zu einer Ausführung der gerufenen Methode. Anschließend kann die aktive Ereignisroutine die spezielle Java-Methode *JistAPI.sleep* rufen, wobei sie ein Modellzeitintervall t_i als Parameter übergibt. Alle nun folgenden Aufrufe von Ereignisroutinen führen zwar weiterhin lediglich zu Eintragungen im Ereigniskalender, nun allerdings mit dem Ausführungszeitpunkt $(t_0 + t_i)$. Die aktive Ereignisroutine kann beliebig oft und in beliebiger Reihenfolge *JistAPI.sleep* und verschiedene Ereignisroutinen aufrufen. Da *JistAPI.sleep* allerdings nur positive Parameterwerte erlaubt, müssen alle Eintragungen im Ereigniskalender in chronologischer Reihenfolge bezüglich der Ausführungszeitpunkte vorgenommen werden. Sobald die aktive Ereignisroutine endet, wird analog zu üblichen Simulationskernen aus dem Ereigniskalender die Ereignisroutine mit dem frühesten Ausführungszeitpunkt ermittelt und mit dieser fortgefahren.

Ein Vorteil von JIST und dessen primäre hervorzuhebende Eigenschaft überhaupt ist die hohe Ausführungsgeschwindigkeit der Simulationsläufe, die durch die direkte Verankerung der Simulationsausführungssemantik im Simulationsmodell erreicht wurde.

Auch wenn die Modelle in JIST weitestgehend wie ereignisorientierte Simulationsmodelle wirken, gibt es doch gegenüber letzteren erhebliche Einschränkungen. Die wohl tiefgreifendste ist das Verbot geteilter Referenzen, dessen Definition und Auswirkungen mit dem in Abschnitt 3.6 beschriebenen Verbot geteilter Referenzen in optimistisch-parallelen Simulationsmodellen identisch ist. Der Zwang, innerhalb einer Ereignisroutine neue Ereignisse in der Reihenfolge ihres späteren Eintritts in den Ereigniskalender einzutragen, wurde bereits weiter oben erwähnt.

Diese zwingend sicherzustellenden Eigenschaften stellen zwar auf theoretischer Ebene keine Einschränkungen bei der Modellbildung dar – alle ereignisorientierten Modelle lassen sich prinzipiell in Modelle umwandeln, die die gegebenen Anforderungen erfüllen. In der Praxis ergeben sich daraus jedoch bei vielen Szenarien ernsthafte Probleme bei der Modellierung, da sich die Modelle sowohl strukturell als auch in Semantikdetails weit vom Ausgangsszenario entfernen. Die daraus resultierenden Folgen bezüglich des Aufwandes bei der Entwicklung sowie der späteren Wartung der Simulationsmodelle wurden bereits in der Einleitung diskutiert.

JiST ist 2004 im Rahmen einer Dissertation [Bar04] an der Cornell University entstanden. In der Praxis wird JiST selbst vergleichsweise selten zur Implementation von Simulationen verwendet. Es bildet allerdings die Grundlage für einen ebenfalls an der Cornell University entstandenen Netzwerksimulator namens SWANS, der sich in entsprechenden Nutzerkreisen großer Beliebtheit erfreut (wozu die hohe, durch JiST realisierte Ausführungsgeschwindigkeit wesentlich beigetragen hat). Allerdings wurde die Entwicklung von JiST schon bald nach dem Abschluss der Dissertation seines Autors weitestgehend eingestellt. Die letzte offizielle Version ist vom November 2005. Auch die vom Entwickler angedachte Erweiterung von JiST um die Möglichkeit der parallelen Simulation wurde nie umgesetzt.

ANHANG C

WEITERE UNTERSUCHUNGEN UND ERKENNTNISSE

C.1 Verwendung von JAVAFLOW für sequentielle Prozesse

Nach der erfolgreichen Implementation der optimistisch-parallelen Simulationsbibliothek MYTIMEWARP stellte sich die Frage, ob die dabei eingesetzte Coroutinen/Continuation-Implementation JAVAFLOW auch für die Prozessimplementation in sequentiellen oder konservativ-parallelen Simulationskernen verwendet werden sollte.

Die prinzipielle Eignung von JAVAFLOW in derartigen Simulationskernen ergibt sich bereits aus der Tatsache, dass sequentielle und konservativ-parallele Simulationen als Teilmengen von optimistisch-parallelen Simulationen aufgefasst werden können. Damit bleibt nur die Frage offen, mit welchen Vor- bzw. Nachteilen die Verwendung von JAVAFLOW gegenüber alternativen Coroutinen/Continuation-Implementationen verbunden ist.

Einschränkungen der Prozessimplementation

Wie in Abschnitt 4.6 beschrieben wurde, basieren die meisten in Java implementierten, sequentiellen Simulationsbibliotheken, die prozessorientierte Simulationsmodelle unterstützen, auf dem Coroutinen-Implementationsansatz der zwangsserialisierten Threads. Dieser Ansatz besitzt mehrere Vorteile. Insbesondere löst er gleichzeitig sowohl das Problem der Ablaufsteuerung als auch der Kontextsicherung, da die Thread-Implementation diese Aufgaben übernimmt, und ist dadurch mit vergleichsweise geringem Aufwand umsetzbar.

Demgegenüber besitzt dieser Ansatz jedoch auch einen wesentlichen Nachteil. Bedingt durch die Verlagerung der wesentlichen Coroutinenaufgaben in eine be-

reits existierende Thread-Implementation werden automatisch alle Einschränkungen dieser übernommen. Dies betrifft vor allem die maximale Anzahl gleichzeitig existierender Threads. Im Implementationsansatz der zwangsserialisierten Threads wird diese Anzahl zur maximalen Anzahl gleichzeitig aktiver Coroutinen bzw. in einer entsprechenden Simulationsimplementation zur maximalen Anzahl gleichzeitig existierender Prozesse.

Dabei hängt im Fall der Thread-Implementation von Java die maximale Anzahl möglicher Threads primär von zwei Faktoren ab: der Menge des Speichers, der der JVM vom Betriebssystem zur Verfügung gestellt wird und der Aufteilung dieses Speichers durch die JVM. Vereinfacht betrachtet unterteilt die JVM den Speicher in zwei separate Bereiche:[1]

- den Heap, hier werden zur Laufzeit alle instantiierten Objekte und Felder abgelegt und
- den Stackbereich, in dem die Stacks der einzelnen Threads gespeichert werden.

Kritisch für die maximale Anzahl von Threads ist vor allem der letztgenannte Bereich, da jede Erzeugung eines Threads mit der Anlage eines neuen Stacks verbunden ist. Dabei wird für jeden Stack sofort der maximale Speicherbereich alloziert, den dieser ausfüllen darf. Die konstante Größe dieser Stacks kann im Fall der weit verbreiteten JVM *HotSpot* von Sun Microsystems per Kommandozeilenparameter (*-Xss*) festgelegt werden. Die Größe des gesamten Stackbereichs lässt sich hingegen nur indirekt durch die Festlegung der Größe des Heaps beeinflussen. Bei der *HotSpot*-JVM erfolgt dies durch zwei Kommandozeilenparameter (*-Xms* und *-Xmx*), die die initiale und die maximale Größe des Heaps festlegen.

Um zumindest eine grobe Vorstellung von den maximal möglichen Anzahlen von Threads zu erhalten, wurde ein Testprogramm implementiert. Dieses instantiiert und startet in einer Endlosschleife Threads, deren einzige Aufgabe in der unmittelbaren Selbstunterbrechung bis zum Programmende besteht. Dieses Testprogramm wurde wiederholt auf *olymp* ausgeführt. Dabei wurden sowohl die Größe des Heaps[2] als auch die Gesamtspeichermenge, die der JVM zur Verfügung steht, variiert. Die ermittelten maximalen Anzahlen von Threads sind in der folgenden Tabelle zu sehen:

	Gesamtspeicher				
Heap-Größe	1 GB	2 GB	4 GB	8 GB	16 GB
0,5 GB	328	1333	3353	7374	15418
1 GB	—	830	2844	6867	14912
2 GB	—	—	1832	5851	13893
4 GB	—	—	—	3850	11863
8 GB	—	—	—	—	7857

[1] Auf Details wie die *Method Area*, die je nach JVM separat oder aber Teil des Heaps sein kann sowie die genaue Aufteilung des Heaps soll an dieser Stelle nicht näher eingegangen werden.

[2] Initial- und Maximalgröße wurden stets auf den gleichen Wert gesetzt.

Bei Verwendung von JAVAFLOW besteht die einzige vergleichbare Einschränkung in der Anzahl gleichzeitig gespeicherter Continuations. Da diese auf dem Heap angelegt werden, wird die maximal mögliche Anzahl im Wesentlichen durch dessen Größe bestimmt. Ein Vergleich mit den vorher ermittelten maximalen Anzahlen von Threads ist jedoch aus mehreren Gründen problematisch. Zum einen belegen Continuations im Gegensatz zu Thread-Stacks nur den wirklich benötigten Speicher, was bei simplen Coroutinenaufgaben wie der des Testprogrammes zu sehr kleinen Continuations und daraus resultierend zu unrealistisch großen möglichen Anzahlen selbiger führt. Zum anderen unterliegen sie auf dem Heap der automatischen Speicherfreigabe (garbage collection) der JVM, d. h., nicht mehr benötigte Continuations werden automatisch entfernt.

Im Bewusstsein der begrenzten Aussagekraft der zu erwartenden Ergebnisse wurde das Testprogramm auf Basis von JAVAFLOW reimplementiert, wobei die Continuations der unterbrochenen Coroutinen zusätzlich in einer verketteten Liste gespeichert wurden, damit sie nicht durch die automatische Speicherfreigabe entfernt werden. Ein Testlauf auf einer JVM, die auf 1 GB Speicher, davon 512 MB Heap beschränkt war, wurde erst bei 16.342.572 gestarteten Coroutinen abgebrochen. Allerdings war die JVM bereits ab ca. 15.000.000 Coroutinen ob des knapp werdenden Heap-Speichers fast ausschließlich mit der Speicherbereinigung beschäftigt.

Auch wenn die ermittelten maximalen Thread-Anzahlen bzw. die im Simulationskontext daraus resultierenden möglichen Anzahlen existierender Prozesse für zahlreiche Simulationsszenarien mehr als ausreichend sind, können Implementationen prozessreicher und gleichzeitig speicherintensiver Simulationsmodelle an dieser Grenze scheitern. Insofern erscheint der Einsatz eines alternativen Coroutinen-Implementationsansatzes wie der auf Basis von JAVAFLOW in einem sequentiellen Simulationskern mindestens überlegenswert.

Laufzeitverhalten

Eine weitere Untersuchung beschäftigte sich mit der Frage des Laufzeitverhaltens der Coroutinen-Implementation von JAVAFLOW gegenüber der auf Thread-Basis. Zu diesem Zweck wurde ein weiteres Testprogramm erstellt, dass n Coroutinen generiert, deren Aufgabe darin besteht, sich nach jeder Aktivierung sofort selbst zu unterbrechen und auf die nächste Aktivierung zu warten. Die Coroutinen wurden dabei so implementiert, dass ihre konkrete Umsetzung durch JAVAFLOW oder mittels Threads erst zur Laufzeit durch das Testprogramm festgelegt wird. Das Testprogramm startet nacheinander alle n Coroutinen und beginnt dann in einer Schleife m-mal alle n Coroutinen nacheinander zu aktivieren.

In den vier Diagrammen in Abbildung C.1 sind die jeweils gemessenen Laufzeiten in Abhängigkeit von der Anzahl verwendeter Coroutinen und der Anzahl der Aktivierungen der einzelnen Coroutinen zu sehen.

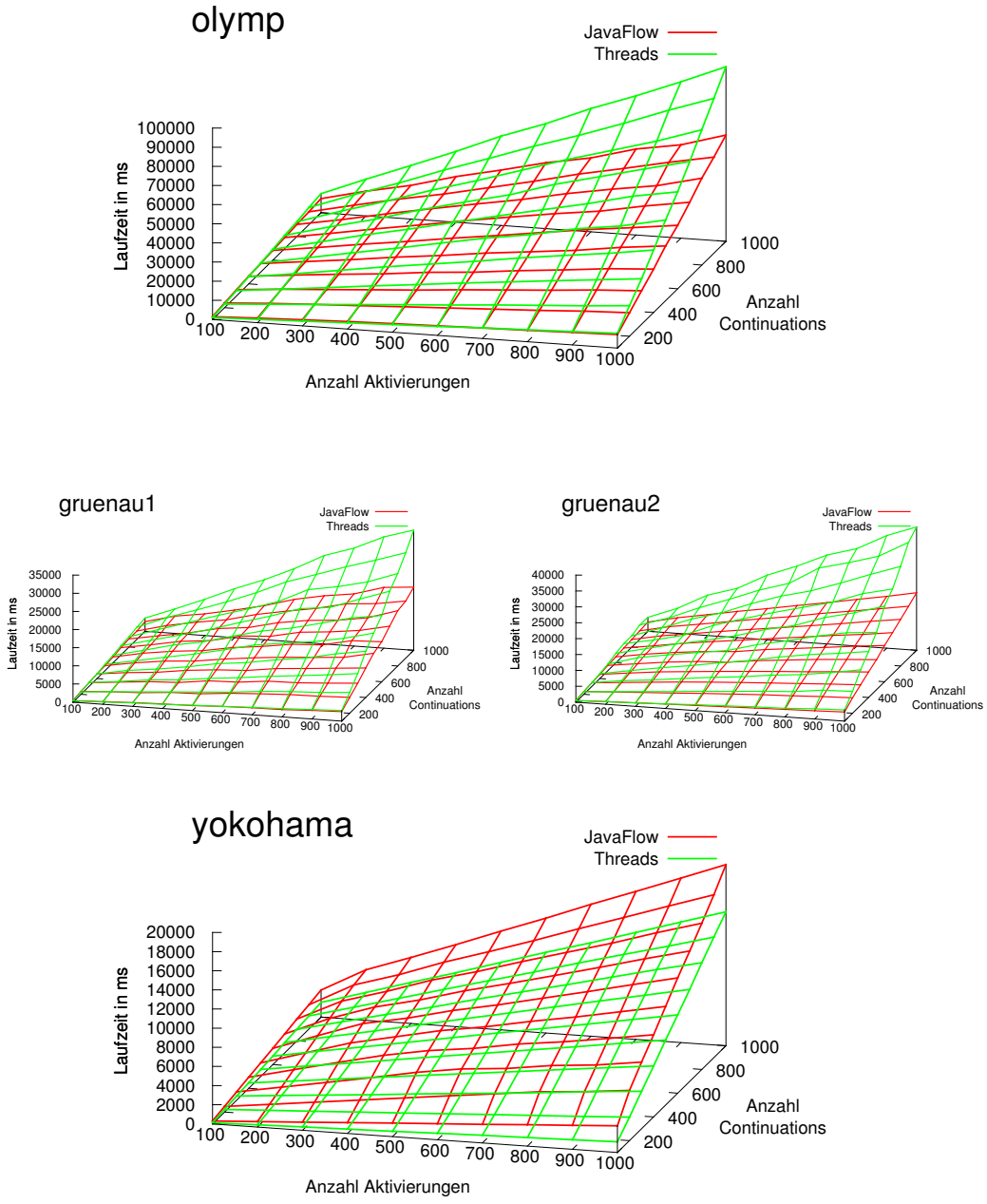

Abbildung C.1: Vergleich der beiden Coroutinen-Implementationen (Thread-basiert, JAVAFLOW) auf *olymp*, *gruenau1*, *gruenau2* und *yokohama*

Das erste Diagramm stellt die Ergebnisse auf *olymp* dar, einem Rechner auf UltraSPARC-Basis[3], der unter dem Betriebssystem Sun Solaris betrieben wird. Auf diesem Rechner war die Coroutinen-Implementation auf Basis von JAVAFLOW der Thread-basierten stets überlegen. Die höhere Geschwindigkeit der Coroutinen-Implementation von JAVAFLOW war auch bei den Testläufen auf den Rechnern *gruenau1* und *gruenau2* zu beobachten. Die ermittelten Laufzeiten sind in den beiden mittleren Diagrammen in Abbildung C.1 zu sehen. Dabei unterscheiden sich die beiden Rechner auf AMD-Opteron-Basis nur im verwendeten Betriebssystem. Während *gruenau1* unter Solaris betrieben wird, kommt auf *gruenau2* ein Linux zum Einsatz.

Für eine Überraschung sorgten jedoch die Testläufe auf *yokohama*. Wie man dem letzten Diagramm in Abbildung C.1 entnehmen kann, war bei den Experimenten auf diesem Rechner die Thread-basierte Variante stets die schnellere. Da *yokohama* mit einem Intel-Core2-Prozessor auf der gleichen x86-Basis arbeitet wie *gruenau1* und *gruenau2* muss der Grund für dieses Ausnahmeverhalten in der Kombination von JVM und Betriebssystem, in diesem Fall Windows, liegen. Stichprobenartige Tests auf weiteren Windows-Rechnern bestätigten diese Annahme.

Zusammenfassung

Vergleicht man abschließend die Eignung der betrachteten Coroutinen-Implementationsansätze für den Einsatz in sequentiellen Simulationskernen, so lässt sich für keine von beiden eine uneingeschränkte Empfehlung geben. Bezüglich der Ausführungsgeschwindigkeit ist der ideale Implementationsansatz vor allem vom verwendeten Betriebssystem abhängig. Betrachtet man den Aufwand einer konkreten Implementation, so ist dieser bei der Thread-Variante deutlich geringer als bei JAVAFLOW (siehe auch Abschnitt 4.6). Dafür erlaubt eine auf JAVAFLOW basierende Simulationsimplementation eine höhere Anzahl von Prozessen zur Simulationslaufzeit.

Der Idealfall wäre eine Implementation beider Ansätze in einem Simulationskern, wobei der konkret verwendete erst zur Laufzeit durch eine Parameterbelegung bestimmt wird. Dadurch kann nicht nur bei Bedarf auf den jeweils geeigneteren Ansatz zurückgegriffen werden. Es wäre vor allem auch möglich, die in dieser Arbeit beobachteten Vor- und Nachteile der beiden Implementationsansätze anhand realer Simulationsmodelle zu überprüfen.

C.2 Laufzeitmessungen in Java und die Fibonaccizahlenberechnung

Ursprünglich sollte der in Abschnitt 6.5 durchgeführte Vergleich zwischen den Simulationsbibliotheken MYTIMEWARP und DESMO-J ebenfalls auf Basis des in den

[3] Die Anzahl der Prozessoren bzw. Prozessorkerne war bei allen in diesem Abschnitt beschriebenen Experimenten irrelevant, da selbst bei der Thread-basierten Coroutinen-Implementation immer nur ein Thread gleichzeitig aktiv war.

Abschnitten 6.3 und 6.4 verwendeten Simulationsmodells der Philosophen mit Fibonaccizahlenberechnung durchgeführt werden. Zu diesem Zweck wurde das Philosophenszenario mit Fibonaccizahlenberechnung erneut auf Basis von DESMO-J implementiert.

Allerdings musste schon nach wenigen Simulationsexperimenten festgestellt werden, dass zumindest für diesen beabsichtigten Vergleich die Fibonaccizahlenberechnung als künstliche Verzögerung ungeeignet ist. Zwar ergab sich auch bei den Experimenten mit der DESMO-J-Implementation der erwartete exponentielle Zusammenhang zwischen Laufzeit und Verzögerungsparameterbelegung, allerdings war der Abhängigkeitsgraph der Simulationslaufzeit von der Verzögerungsparameterbelegung gegenüber allen Ergebnissen der Implementation mit MYTIMEWARP nicht nur nach unten sondern auch deutlich nach rechts verschoben. Dies bedeutet jedoch, dass die Simulationsimplementation mit DESMO-J nicht nur schneller ist als die MYTIMEWARP-Implementation (was selbst bei Verwendung von acht LPs galt!), sondern auch, dass es der DESMO-J-Implementation möglich ist, den exponentiellen Anstieg des Rechenaufwandes „herauszuzögern“.

Verschiedene Tests mit immer weiter vereinfachten Simulationsmodellen bis hin zum Extremfall eines völlig ungekoppelten Modells (ausschließlich denkende Philosophen ohne Gabeln) sowohl auf DESMO-J- als auch MYTIMEWARP-Seite bestätigten schließlich, dass die Ursache in der Laufzeitoptimierung der JVM liegt. Diese beschleunigt die rekursive Fibonaccizahlenberechnung durch das Zwischenspeichern der Ergebnisse der ständig wiederkehrenden, seiteneffektfreien Funktionsaufrufe. Ob und wie stark diese Optimierung durchgeführt wird bzw. werden kann, hängt dabei maßgeblich vom restlichen Programm ab.

Die Funktionsweise von MYTIMEWARP besitzt gleich zwei Eigenschaften, die in diesem Fall eine dynamische Optimierung durch die JVM erschweren: das Verteilen der Berechnungen auf mehrere Threads und vor allem die durch Bytecode-Rewriting umgesetzte Kapselung von Laufzeitzuständen in Continuations. Demgegenüber kann bei der DESMO-J-Implementation, bei der alle Berechnungen in einem einzelnen Thread stattfinden, das Optimierungspotential deutlich besser erkannt und ausgenutzt werden.

Nun war es nicht das Ziel der Arbeit, die (Nicht-)Eignung einer Simulationsbibliothek für die Laufzeitoptimierung einer konkreten und in diesem Fall absichtlich ineffizient implementierten Berechnung zu bewerten. Daher wurde für den angestrebten Vergleich in Abschnitt 6.5 eine alternative künstliche Verzögerung in Form der mehrfachen MD5-Summenberechnung gewählt, deren Laufzeit durch die Optimierung der JVM prinzipiell nicht beeinflusst werden kann.

ANHANG D

VERWENDETE HARDWARE

Im Folgenden werden die wichtigsten Daten der Hardware, die für die in Kapitel 6 bzw. Anhang C beschriebenen Untersuchungen verwendet wurde, zusammengefasst.

Insgesamt wurden Experimente auf fünf verschiedenen Rechnern (*olymp*, *mnemosyne*, *gruenau1*, *gruenau2* und *yokohama*) durchgeführt. Dabei spielte *olymp* eine besondere Rolle, da es auf diesem Mehrprozessorsystem möglich war, für jeden Testlauf eine Menge von Prozessoren exklusiv zu reservieren, d. h., weder die Programme anderer Nutzer noch die Betriebsystemprozesse konnten die vorgenommenen Messungen beeinflussen.

olymp

System	Sun Fire V1280
Hersteller	Sun Microsystems, Inc.
Anzahl Prozessoren	12
Anzahl Prozessorkerne	12
Prozessortyp	1,2 GHz UltraSPARC III
Hauptspeicher	24 GB
Betriebssystem	Sun Solaris 5.10
Java Virtual Machine	JRE 1.6.0

mnemosyne

System	Sun SPARC Enterprise M4000
Hersteller	Sun Microsystems, Inc.
Anzahl Prozessoren	4
Anzahl Prozessorkerne	16
Prozessortyp	2,4 GHz UltraSPARC VII
Hauptspeicher	32 GB
Betriebssystem	Sun Solaris 5.10
Java Virtual Machine	JRE 1.6.0

gruenau1

System	Sun Fire X4600 M2
Hersteller	Sun Microsystems, Inc.
Anzahl Prozessoren	8
Anzahl Prozessorkerne	16
Prozessortyp	2,6 GHz Dual-Core AMD Opteron
Hauptspeicher	32 GB
Betriebssystem	Sun Solaris 5.10
Java Virtual Machine	JRE 1.6.0

gruenau2

System	Sun Fire X4600 M2
Hersteller	Sun Microsystems, Inc.
Anzahl Prozessoren	8
Anzahl Prozessorkerne	16
Prozessortyp	2,6 GHz Dual-Core AMD Opteron
Hauptspeicher	32 GB
Betriebssystem	SUSE Linux Enterprise Server 10 (x86_64)
Java Virtual Machine	JRE 1.6.0

yokohama

System	Dell Optiplex 745
Hersteller	Dell, Inc.
Anzahl Prozessoren	1
Anzahl Prozessorkerne	2
Prozessortyp	2,13 GHz Intel Core2 6400
Hauptspeicher	3 GB
Betriebssystem	Windows XP Service Pack 3
Java Virtual Machine	JRE 1.6.0

STICHWORTVERZEICHNIS

LITERATURVERZEICHNIS

[AGH06] ARNOLD, Ken ; GOSLING, James ; HOLMES, David: *The Java Programming Language, Fourth Edition*. Addison-Wesley, Upper Saddle River, NJ, USA, 2006. – ISBN 0–321–34980–6

[Apaa] APACHE SOFTWARE FOUNDATION: *Apache Commons Homepage*. `commons.apache.org`, letzter Abruf: Juni 2010

[Apab] APACHE SOFTWARE FOUNDATION: *Byte Code Engineering Library (BCEL) Homepage*. `jakarta.apache.org/bcel`, letzter Abruf: Juni 2010

[Apac] APACHE SOFTWARE FOUNDATION: *JavaFlow Homepage*. `commons.apache.org/sandbox/javaflow`, letzter Abruf: Juni 2010

[Ash56] ASHBY, W. Ross: *An introduction to cybernetics*. Chapman & Hall, London, UK, 1956

[Bar04] BARR, Rimon: *An efficient, unifying apporach to simulation using virtual machines*. Cornell University, Ithaca, NY, USA, Dissertation, 2004

[BB97] BOOTH, Chris J. M. ; BRUCE, Donald I.: Stack-free process-oriented simulation. In: *PADS '97: Proceedings of the eleventh workshop on Parallel and distributed simulation*. IEEE Computer Society, Washington, DC, USA, 1997, S. 182–185. – ISBN 0–8186–7965–4

[BDO85] BILES, William E. ; DANIELS, Cheryl M. ; O'DONNELL, Tamilea J.: Statistical considerations in simulation on a network of microcomputers. In: *WSC '85: Proceedings of the 17th conference on Winter simulation*. ACM, New York, NY, USA, 1985, S. 388–393. – ISBN 0–911801–07–3

[Bev07] BEVIN, Geert: *Continuations in Java*. Leading Edge Java. März 2007. `www.artima.com/lejava/articles/continuations.html`, letzter Abruf: Juni 2010

[Bev08] BEVIN, Geert: *JSR Continuations Provider API – working draft*. 2008. `rifers.org/wiki/display/RIFECNT/JSR+Continuations+Provider+API`, letzter Abruf: Juni 2010

[Bir81] BIRTWISTLE, Graham: Introduction to Demos. In: *WSC '81: Proceedings of the 13th conference on Winter simulation*. IEEE Press, Piscataway, NJ, USA, 1981, S. 559–572

[BLC02] BRUNETON, Eric ; LENGLET, Romain ; COUPAYE, Thierry: ASM: A code manipulation tool to implement adaptable systems. In: *Adaptable and extensible component systems*, 2002

[Cel91] CELLIER, Francois E.: *Continuous System Modeling*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1991. – ISBN 0–387–97502–0

[CM81] CHANDY, K. M. ; MISRA, J.: Asynchronous distributed simulation via a sequence of parallel computations. In: *Communications of the ACM*, Bd. 24, Nr. 4, 1981, S. 198–206. – ISSN 0001–0782

[Com84] COMFORT, John C.: The simulation of a master-slave event set processor. In: *Simulation*, Bd. 42, Nr. 3, 1984, S. 117–124

[Con63] CONWAY, Melvin E.: Design of a separable transition-diagram compiler. In: *Communications of the ACM*, Bd. 6, Nr. 7, 1963, S. 396–408. – ISSN 0001–0782

[DDH72] DAHL, O. J. (Hrsg.) ; DIJKSTRA, E. W. (Hrsg.) ; HOARE, C. A. R. (Hrsg.): *Structured programming*. Academic Press Ltd., London, UK, 1972. – ISBN 0–12–200550–3

[Dij71] DIJKSTRA, Edsger W.: Hierarchical Ordering of Sequential Processes. In: *Acta Informatica*, Bd. 1, 1971, S. 115–138

[DN66] DAHL, Ole-Johan ; NYGAARD, Kristen: SIMULA: an ALGOL-based simulation language. In: *Communications of the ACM*, Bd. 9, Nr. 9, 1966, S. 671–678. – ISSN 0001–0782

[Eve06] EVESLAGE, Ingmar: *Reimplementation einer Stahlwerkssimulation auf der Basis der Simulationsbibliothek ODEMx*. Humboldt-Universität zu Berlin, Studienarbeit, August 2006

[FA96] FISCHER, Joachim ; AHRENS, Klaus: *Objektorientierte Prozeßsimulation in C++*. Addison-Wesley, Bonn, 1996

[Fis82] FISCHER, Joachim: Modellierung und Simulation paralleler diskreter, kontinuierlicher und kombinierter Prozesse in SIMULA. In: *ZfR-Informationen*, Nr. ZfR-82.20, Juli 1982

[Fuj93] FUJIMOTO, Richard M.: Feature Article - Parallel Discrete Event Simulation: Will the Field Survive? In: *INFORMS Journal on Computing*, Bd. 5, Nr. 3, 1993, S. 213–230

[Fuj99] FUJIMOTO, Richard M.: *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1999. – ISBN 0–471–18383–0

[Gai80] GAINES, Brian: General Systems Research: Quo Vadis. In: *Ed. General Systems* Bd. 24, Society for General Systems Research, 1980, S. 1–9

[Ger03] GERSTENBERGER, Ralf: *ODEMx – Neue Lösungen für die Realisierung von C++-Bibliotheken zur Prozesssimulation*. Humboldt-Universität zu Berlin, Diplomarbeit, 2003

[GG96] GRISWOLD, Ralph E. ; GRISWOLD, Madge T.: *The ICON Programming Language*. Annabooks, 1996. – ISBN 1–573–98001–3

[Gle10] GLEIM, Urs: Handwerkszeug – Grundlagen der parallelen Programmierung. In: *iX*, Nr. 5, März 2010

[GP01] GEHLSEN, Björn ; PAGE, Bernd: A framework for distributed simulation optimization. In: *WSC '01: Proceedings of the 33nd conference on Winter simulation*. IEEE Computer Society, Washington, DC, USA, 2001, S. 508–514. – ISBN 0–7803–7309–X

[HDA94] HIEB, Robert ; DYBVIG, R. Kent ; ANDERSON, Claude W.: Subcontinuations. In: *LISP and Symbolic Computation*, Bd. 7, Nr. 1, 1994, S. 83–110. – ISSN 0892–4635

[Hel00] HELSGAUN, Keld: Discrete Event Simulation in Java. In: *Datalogiske Skrifter (Writings on Computer Science)*, 2000

[Hel01] HELSGAUN, Keld: jDisco – a Java framework for combined discrete and continuous simulation. In: *Datalogiske Skrifter (Writings on Computer Science)*, 2001

[Hel07] HELBING, Chris: *Simulation eines Stahlwerksabschnittes mit der Simulationsbibliothek DESMO-J*. Humboldt-Universität zu Berlin, Studienarbeit, März 2007

[Hül06] HÜLSENBUSCH, Ralph: Krieg der Kerne. In: *iX*, Nr. 4, April 2006

[JBW+87] JEFFERSON, D. ; BECKMAN, B. ; WIELAND, F. ; BLUME, L. ; DILORETO, M.: Time warp operating system. In: *SIGOPS Operating Systems Review*, Bd. 21, Nr. 5, 1987, S. 77–93. – ISSN 0163–5980

[Jef85] JEFFERSON, David R.: Virtual time. In: *ACM Transactions on Programming Languages and Systems*, Bd. 7, Nr. 3, 1985, S. 404–425. – ISSN 0164–0925

[JLV02] JACOBS, Peter H. M. ; LANG, Niels A. ; VERBRAECK, Alexander: D-SOL; a distributed Java based discrete event simulation architecture. In: *WSC '02: Proceedings of the 34th conference on Winter simulation*, 2002, S. 793–800. – ISBN 0–7803–7615–3

[JS82] JEFFERSON, David ; SOWIZRAL, Henry: Fast Concurrent Simulation Using the Time Warp Mechanism, Part 1: Local Control. / The Rand Corporation. Santa Monica, CA, USA, Dezember 1982. – Forschungsbericht

[Kar77] KARPLUS, Walter J.: The spectrum of mathematical modeling and systems simulation. In: *SIGSIM Simulation Digest*, Bd. 9, Nr. 1, 1977, S. 32–38. – ISSN 0163–6103

[Knu68] KNUTH, Donald E.: *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, Reading, MA, USA, 1968

[Kun08] KUNERT, Andreas: Optimistic-parallel process-oriented DES in Java using Bytecode Rewriting. In: *Proceedings of MESM'2008*, Eurosis, 2008, S. 15–21

[KW78] KORN, Granino ; WAIT, John: *Digital continuous system simulation*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1978

[LP99] LECHLER, Tim ; PAGE, Bernd: DESMO-J: An Object Oriented Discrete Simulation Framework in Java. In: *Proceedings of the 11th European Simulation Symposium*. SCS Publishing House, Erlangen, 1999

[LY99] LINDHOLM, Tim ; YELLIN, Frank: *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, Reading, MA, USA, 1999. – ISBN 0–201–43294–3

[Mar80] MARLIN, C.: Coroutines: A Programming Methodology, a Language Design and an Implementation. In: *Lecture Notes in Computer Science*, Bd. 95, 1980

[MD97] MEYER, Jon ; DOWNING, Troy: *Java virtual machine*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997. – ISBN 1–56592–194–1

[Meh94] MEHL, Horst: *Methoden verteilter Simulation*. Vieweg, Braunschweig, 1994

[MI04] MOURA, Ana L. ; IERUSALIMSCHY, Roberto: Revisiting Coroutines / PUC-Rio. Rio de Janeiro, Brasilien, Juni 2004. – Forschungsbericht

[Min65] MINSKY, Marvin: Matter, mind and models. In: *Proceedings IFIP Congress* Bd. 1, 1965, S. 45–49

[Mis86] MISRA, Jayadev: Distributed discrete-event simulation. In: *ACM Computing Surveys*, Bd. 18, Nr. 1, 1986, S. 39–65. – ISSN 0360–0300

[MPN93] MADSEN, Ole L. ; PEDERSEN, Birger M. ; NYGAARD, Kristen: *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. – ISBN 0–201–62430–3

[MR] MEYER, Jonathan ; REYNAUD, Daniel: *Jasmin Homepage*. `jasmin.sourceforge.net`, letzter Abruf: Juni 2010

[Nic97] NICOL, D. M.: Parallel Discrete Event Simulation: So who cares? (Keynote presentation). In: *PADS'97: 11th workshop on Parallel and distributed simulation*, 1997

[ORCAE06] ORTEGA-RUIZ, Jose A. ; CURDT, Torsten ; AMETLLER-ESQUERRA, Joan: Continuation-based Mobile Agent Migration / University of Barcelona. 2006. – Forschungsbericht

[OW2] OW2 CONSORTIUM: *ASM Homepage*. `asm.objectweb.org`, letzter Abruf: Juni 2010

[Pet99] PETERS, Tim: *Fake threads*. Python-Entwickler-Forum. Juli 1999. `mail.python.org/pipermail/python-dev/1999-July/000467.html`, letzter Abruf: Juni 2010

[PF98] PERUMALLA, Kalyan S. ; FUJIMOTO, Richard M.: Efficient large-scale process-oriented parallel simulations. In: *WSC'98: Proceedings of the 30th conference on Winter simulation*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, S. 459–466. – ISBN 0–7803–5134–7

[PK05] PAGE, Bernd ; KREUTZER, Wolfgang: *The Java Simulation Handbook*. Shaker Verlag, Aachen, 2005. – ISBN 3–8322–3771–2

[Pre90] PREISS, B.: Performance of Discrete Event Simulation on a Multiprocessor Using Optimistic and Conservative Synchronization. In: *Proceedings of the International Conference on Parallel Processing* Bd. 3, 1990, S. 218–222

[Que03] QUEINNEC, Christian: Inverting back the inversion of control or, continuations versus page-centric programming. In: *SIGPLAN Notices*, Bd. 38, Nr. 2, 2003, S. 57–64. – ISSN 0362–1340

[Que04] QUEINNEC, Christian: Continuations and Web Servers. In: *Higher Order and Symbolic Computation*, Bd. 17, Nr. 4, 2004, S. 277–295. – ISSN 1388–3690

[Rey] REYNAUD, Daniel: *Tinapoc (Tinapoc is not another pun on coffee) Homepage*. `tinapoc.sourceforge.net`, letzter Abruf: Juni 2010

[RIF] RIFE TEAM: *Rife Homepage*. `www.rifers.org`, letzter Abruf: Juni 2010

[Roe] ROEDER, Harald: *ClassFileAnalyzer (Can) Homepage*. `classfileanalyzer.javaseiten.de`, letzter Abruf: Juni 2010

[Sch90] SCHWARZE, Gunter: *Digitale Simulation*. Akademie-Verlag, Berlin, 1990

[Sut05] SUTTER, Herb: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. In: *Dr. Dobb's Journal*, Bd. 30, Nr. 3, 2005, S. 202–210

[Tat00] TATHAM, Simon: *Coroutines in C*. 2000. `www.chiark.greenend.org.uk/~sgtatham/coroutines.html`, letzter Abruf: Juni 2010

[Tat06] TATE, Bruce: *Crossing borders: Continuations, Web development, and Java programming*. IBM developerWorks series: Java theory and practice. März 2006. `www-128.ibm.com/developerworks/java/library/j-cb03216/`, letzter Abruf: Juni 2010

[TN02] TEO, Yong M. ; NG, Yew K.: SPaDES/Java: Object-Oriented Parallel Discrete-Event Simulation. In: *SS '02: Proceedings of the 35th Annual Simulation Symposium*. IEEE Computer Society, Washington, DC, USA, 2002, S. 245

[TNO02] TEO, Y. M. ; NG, Y. K. ; ONGGO, B. S. S.: Conservative simulation using distributed-shared memory. In: *PADS'02: Proceedings of the sixteenth workshop on Parallel and distributed simulation*. IEEE Computer Society, Washington, DC, USA, 2002, S. 3–10. – ISBN 0–7695–1608–4

[Vak92] VAKILI, Pirooz: Massively parallel and distributed simulation of a class of discrete event systems: a different perspective. In: *ACM Transactions on Modeling and Computer Simulation*, Bd. 2, Nr. 3, 1992, S. 214–238. – ISSN 1049–3301

[War06] WARTALA, Ramon: Sandförmchen – Apaches Jakarta Commons Sandbox. In: *iX*, Nr. 4, April 2006

[Wil86] WILSON, Andrew: Parallelization of an event driven simulator for computer systems simulation. In: *Simulation*, Bd. 49, Nr. 2, 1986, S. 72–78

[Wir83] WIRTH, N.: *Programming in Modula-2*. Springer-Verlag TELOS, Santa Clara, CA, USA, 1983. – ISBN 0–540–15078–1

[WSY83] WYATT, Dana L. ; SHEPPARD, Sallie ; YOUNG, Robert E.: An experiment in microprocessor-based distributed digital simulation. In: *WSC'83:*

Proceedings of the 15th conference on Winter simulation. IEEE Press, Piscataway, NJ, USA, 1983, S. 271–278

DANKSAGUNG

Mein erster Dank gilt meinem Betreuer, *Prof. Dr. Joachim Fischer*, sowohl für die Unterstützung bei der Suche und Konkretisierung meines Dissertationsthemas als auch für seine wertvolle Hilfe in allen Phasen der Erstellung der vorliegenden Arbeit. Insbesondere bin ich ihm für seine zahlreichen kritischen Kommentare dankbar.

Des Weiteren bin ich meinen ehemaligen Kollegen am Lehrstuhl Systemanalyse zu Dank verpflichtet, vor allem *Falko Theisselmann*, *Frank Kühnlenz*, *Daniel Sadilek*, *Dr. Klaus Ahrens* und *Ingmar Eveslage*. Ihnen verdanke ich zahlreiche fruchtbare und teilweise äußerst lebhafte Diskussionen sowie wertvolle Hinweise sowohl in den Konzeptions- und Implementationsphasen von MyTimeWarp als auch während des Schreibens der vorliegenden Arbeit. Außerdem möchte ich mich bei ihnen entschuldigen, dass ich den damalig stärksten Rechner des Lehrstuhls (*olymp*) zeitweise so exzessiv für meine Experimente beansprucht habe.

Auch bei meinen neuen Kollegen am Rechenzentrum (CMS) der Humboldt-Universität zu Berlin möchte ich mich bedanken, insbesondere bei *Dr. Jens Doebler* und *Jana Kunze* für ihre wertvolle Hilfe in der Endphase der Dissertationsschrift. Des Weiteren bedanke ich mich bei *Michael Rybczak* für die Möglichkeit, meine Simulationsexperimente auch auf dem neuen zentralen Datenbankserver der Humboldt-Universität (*mnemosyne*) durchführen zu dürfen. Außerdem bin ich *Dr. Günther Kroß* und *Daniel Rohde* dafür zu Dank verpflichtet, dass sie mir trotz meiner Einarbeitungsphase beim CMS ermöglicht haben, dem Abschluss meiner Dissertation höchste Priorität zu geben.

Besonders muss ich mich bei meinen Lektoren *Uwe Düffert*, *Dr. Jana Schütze*, *Dr. Stefan Kirchner* und *Dr. Jan-Peter Bell* bedanken, die die gesamte Arbeit mit einer Akribie studiert und korrigiert haben, die weit über meine Erwartungen hinausging.

Schließlich möchte ich mich auch bei meinen Eltern bedanken, die in dem gesamten Promotionsvorhaben eine nicht zu vernachlässigende, treibende Kraft darstellten.

Mein größter Dank gilt jedoch meiner Frau Vanessa. Ohne ihre ständige, uneingeschränkte Unterstützung wäre die vorliegende Arbeit wohl nie zu einem erfolgreichen Abschluss gekommen.

SELBSTÄNDIGKEITSERKLÄRUNG

Hiermit erkläre ich, dass ich die vorliegende Dissertationsschrift „Prozessorientierte optimistisch-parallele Simulation“ selbständig und nur unter Zuhilfenahme der angegebenen Quellen und Hilfsmittel angefertigt habe.

Des Weiteren bestätige ich, dass ich noch keinen Doktorgrad besitze und mich auch nicht bereits anderweitig um einen solchen beworben habe.

Die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin gemäß dem Amtlichen Mitteilungsblatt Nr. 34/2006 ist mir bekannt.

Berlin, den 20. Juli 2010

Andreas Kunert